The ePNK: A generic PNML tool Users' and Developers' Guide for Version 1.0.0

Kindler, Ekkart

Publication date:
2012

Document Version
Publisher's PDF, also known as Version of record

The ePNK: A generic PNML tool
Users' and Developers' Guide
for Version 1.0.0

Ekkart Kindler
Technical University of Denmark
DTU Informatics
DK-2800 Kgs. Lyngby
Denmark
eki@imm.dtu.dk

December 2012

IMM-Technical Report-2012-14
(revised and extended version of IMM-Technical Report-2011-03)

DTU Informatics
Department of Informatics and Mathematical Modeling
Technical University of Denmark

Building 321, DK-2800 Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-Technical Reports: ISSN 1601-2321

Abstract

The *Petri Net Markup Language* (*PNML*) is an XML based interchange format for all kinds of Petri nets, which was published as International Standard ISO/IEC 15909-2 in February 2011. Technically, ISO/IEC 15909-2 is defining an interchange format for three different kinds of high-level Petri nets and a simple version of Place/Transition systems only. But, one of the objectives of PNML was to provide a means for exchanging any kind of Petri net [10, 29, 1]. To this end, the concept of a *Petri Net Type Definition* (*PNTD*) was introduced, which is subject of a newly issued standardisation project: ISO/IEC 15909-3.

There are many tools supporting one form of PNML or the other, and, in particular, there is the PNML Framework [7], which helps tool developers to ease the implementation of PNML by providing a framework and an API for loading and saving Petri net documents in PNML. This framework is based on the *Eclipse Modeling Framework* (*EMF*) [2] and has its focus on the underlying meta-models of Petri nets. The PNML Framework, however, is not generic in the following sense: Whenever a new Petri net type is created, the code for the complete tool needs to be regenerated. Moreover, the PNML Framework does not come with a graphical editor for Petri nets.

The ePNK overcomes these limitations: It provides an extension-point, so that new Petri net types can be plugged in to the existing tool without touching the code of the ePNK. For defining a new Petri net type, the developer, basically, needs to give a class diagram (actually an Ecore diagram) defining the concepts of the new Petri net type, along with a mapping of these concepts to XML syntax. This type can then be plugged into the ePNK, and the graphical editor of the ePNK will be able to edit nets of this new type with all its features. Likewise, the ePNK allows to plug in new functionality for analysis and verification of Petri nets or any other kind of application for Petri nets.

Actually, this was the idea when we started the development of the *Petri Net Kernel* (*PNK*) about 15 years ago [17, 12, 20]. At that time, however, we had to implement all of the IDE functionality of such a tool ourselves. Today, we can make use of the Eclipse platform [27], which helps us focusing on the Petri net

specific parts; we get all the functionality of a nice IDE, basically, for free. This is why we named the tool *ePNK*: it can be considered to be an *Eclipse-based Petri Net Kernel*. But, it is just the spirit and their idea that the PNK and the ePNK have in common; technically, there is not a single line of code from the PNK in the ePNK, and the ePNK is not compatible with the PNK.

What is more, we use the nice features of EMF, GMF, and Xtext for developing the ePNK in a model-based way. In this way, the complete development process of the ePNK is a case study in model-based software engineering using EMF and related technologies. This, actually, was the driving force behind this project. The evaluation and the lessons learned during this project, however, will be reported at an other occasion and to a different audience. This manual focuses on how to use the ePNK as an end user, and shows how a developer can use the extension mechanisms of the ePNK for providing new Petri net types along with their XML syntax, and how to add new functionality to the ePNK.

A first version of this manual has been published in February 2011 as IMM-Technical Report-2011-03 already, which referred to version 0.9.1 of the ePNK. The current version of this document refers to version 1.0.0 of the ePNK, which was released in October 2012.

Contents

Chapter 1

Installation

This chapter discusses the installation of the ePNK (version 1.0.0). Readers who are interested in getting an idea of what the ePNK is and who do not want to work with the PNK right away can skip this chapter.

1.1 Prerequisites

In order to install the ePNK, you need to have Java 1.6 (or higher) and Eclipse 3.7 (Indigo) or Eclipse 4.2 (Juno) installed on your computer. In this version of the manual, we discuss the installation of the ePNK version 1.0.0 only. For installing other versions of the ePNK or for installing it on other versions of Eclipse, you might find information on the *ePNK installation page*[1].

For the installation of Java, please refer to `http://www.java.com/`.

If you are new to Eclipse, it is recommended that you install the *Eclipse Classic* version. Download this Eclipse version for your operating system from `http://www.eclipse.org/downloads/` and extract the downloaded file to some directory; after the extraction, you will find a folder named "eclipse" in this directory, and in this folder, you will find an executable file also called "eclipse" (e. g. "eclipse.exe" on the Windows platform). Executing this file will start Eclipse.

If you are new to Eclipse, you can get a quick overview of the Eclipse Integrated Development Environment (IDE) at `http://www.vogella.de/articles/Eclipse/article.html`. Once you have installed and started Eclipse, you will find much more information on Eclipse in the "Workbench

[1] `http://www2.imm.dtu.dk/~ekki/projects/ePNK/install-details.html`

User Guide" in the Eclipse help: You can open it via the "Help" menu in the Eclipse toolbar under "Help Contents".

1.2 Installing the ePNK in Eclipse

Once you have installed Eclipse, you can install the ePNK from the Eclipse workbench. To this end, the ePNK is made available via an *Eclipse update site*: `http://www2.imm.dtu.dk/~ekki/projects/ePNK/indigo/update/`

In order to install the ePNK from there to your Eclipse installation, you should proceed as follows (after you have started it and selected a workspace):

1. In the Eclipse toolbar, select "Help" → "Install New Software...", which will open an install dialog.

2. In the install dialog, press the "Add..." button to add a new update site. In the "Add Site" dialog, enter some name (e. g. "ePNK Update Site") and the URL

   `http://www2.imm.dtu.dk/~ekki/projects/ePNK/indigo/update/`

   as location, and then press okay.

3. Now, select the newly created ePNK update site in the still open install dialog. After some time, some ePNK items should pop up in the dialog. From there, you can select the features of the ePNK you want.

   For working with this manual, you should at least select the following features from the category "ePNK Features":

   - ePNK Basic Extensions 1.0.0
   - ePNK Core 1.0.0
   - ePNK HLPNGs 1.0.0
   - ePNK Tutorial 1.0.0

   If you intend to import high-level nets from other tools than the ePNK, it is recommended that you also install the feature

   - ePNK: HLPNG Label Serialisation (experimental) 0.2.0

   from category "ePNK Experimental Projects".

   If you want to simulate high-level nets, you should also select the feature

- ePNK: HLPNG Simulator 0.1.1

from category "HLPNG Simulator".

You will not need the features from the "ECNO Projects" category, which are a project in their own right (see [16] for more information on the ECNO project). Since they are based on the ePNK, they are deployed from the same update site.

4. After you have selected the features you want, make sure that the box "Contact all update sites during install to find required software" is checked; this will guarantee that all additional features from Eclipse that the ePNK requires will be automatically installed together with the ePNK features (EMF, GMF, Xtext, etc.).

   Then press press okay.

5. Follow through the installation process (don't forget to accept the license agreement).

   Note: If you get an error of the kind

   ```
   Cannot complete the install because one
   or more required items could not be found.
   ...
   ```

   you probably forgot to check the box "Contact all update sites during install to find required software" or have selected a wrong combination of features. In that case, go back and select the right combination as explained above.

6. Then, the selected features of the ePNK and all other required features will be installed; it is a good idea to restart Eclipse after that (Eclipse will ask you to do that anyway).

In case you intend to develop new functions and, in particular, new Petri net types for the ePNK, you might want to install the tools necessary for that purpose already now – while at it. You need to install the "EMF Modeling Framework SDK" and the "Ecore Tools SDK" from the standard Eclipse update site. The details are described in Sect. 4.1.2.

Chapter 2

Introduction

This chapter gives a brief overview of the *Petri Net Markup Language* (*PNML*) as well as of the concepts and ideas of the ePNK and its main features. In the end of this chapter, there is some information for different kinds of readers on what to read and on how to read this manual.

2.1 Motivation

The PNML is an XML-based interchange format for all kinds of Petri nets, which allows different tools to exchange Petri net models among each other. One of PNML's main features is that it is generic, which means that it provides a mechanism for defining own types of Petri nets, which are called *Petri net type definitions* (*PNTD*). These Petri net type definitions define the additional concepts of the new Petri net type, as well as the representation of these new concepts in XML syntax. It is also possible that different tools include their *tool specific information* to PNML documents, which is information that can be safely ignored by other tools.

Up to now, there was no tool that fully supported these ideas, in such a way that the tool would allow a developer to define and plug in new Petri net types and additional tool specific extensions. And there was no generic editor supporting all Petri net types, once they are plugged in.

The lack of such a generic tool support was the starting point for developing the *ePNK*.

2.2 The Petri Net Markup Language

In order to better understand the ideas of the ePNK, we briefly discuss the main concepts and ideas of the PNML here. For more information on the PNML and on ISO/IEC 15909-2, we refer to [13, 6] or to the International Standard ISO/IEC 15909-2:2011 itself [8].

2.2.1 The PNML core model

As stated above, extensibility and genericity were two of the main objectives behind the PNML [11]. This is achieved by identifying the concepts that are common to all Petri nets in the so-called *PNML core model*. The common concepts are mainly *places*, *transitions* and *arcs*, and that these *objects* can have some kind of *label*. The PNML core model also provides means for splitting up larger Petri nets into *pages*; connections between nodes on different pages can be established by *reference places* or *reference transitions*. And PNML defines all kinds of graphical information that can be attached to the different elements, such as position, size, font-type, and font-size.

Note that in graphical editors, a label would typically be shown as an annotation attached (close) to the object it belongs to. Labels that should be shown as annotations are, therefore called *Annotations*. Some labels, however, are not supposed to be shown as annotations; for example, if there are different kinds of arcs, the kind or arc might be defined as an attribute of the arc in the properties view of a tool. And in some graphical tools, the graphical representation of the arc itself might change dependent on the value of this attribute; an arc of kind "read", might be graphically represented as a line without any arrow heads or with arrow heads at both ends. An arc of kind "inhibitor" might be shown as a lollipop. Therefore, these kind of labels are called *attributes*.

In addition, the PNML core model defines the possible relation between these elements. In particular, it defines that places and transitions, which are generalized as *nodes*, are contained in pages and that arcs may connect these nodes. Figure 2.1 shows the PNML core model of ISO/IEC 15909-2 as a UML diagram. Note that the PNML core model of the ePNK is slightly more general than the one defined in ISO/IEC 15909-2; this way, it is possible to capture even more variants of Petri nets. For now, these differences are not so important; the most relevant differences are explained later in Sect. 4.2.

Note that there is only one concrete type of label defined in the PNML core model itself, which is the *name* of an element. All the other possible

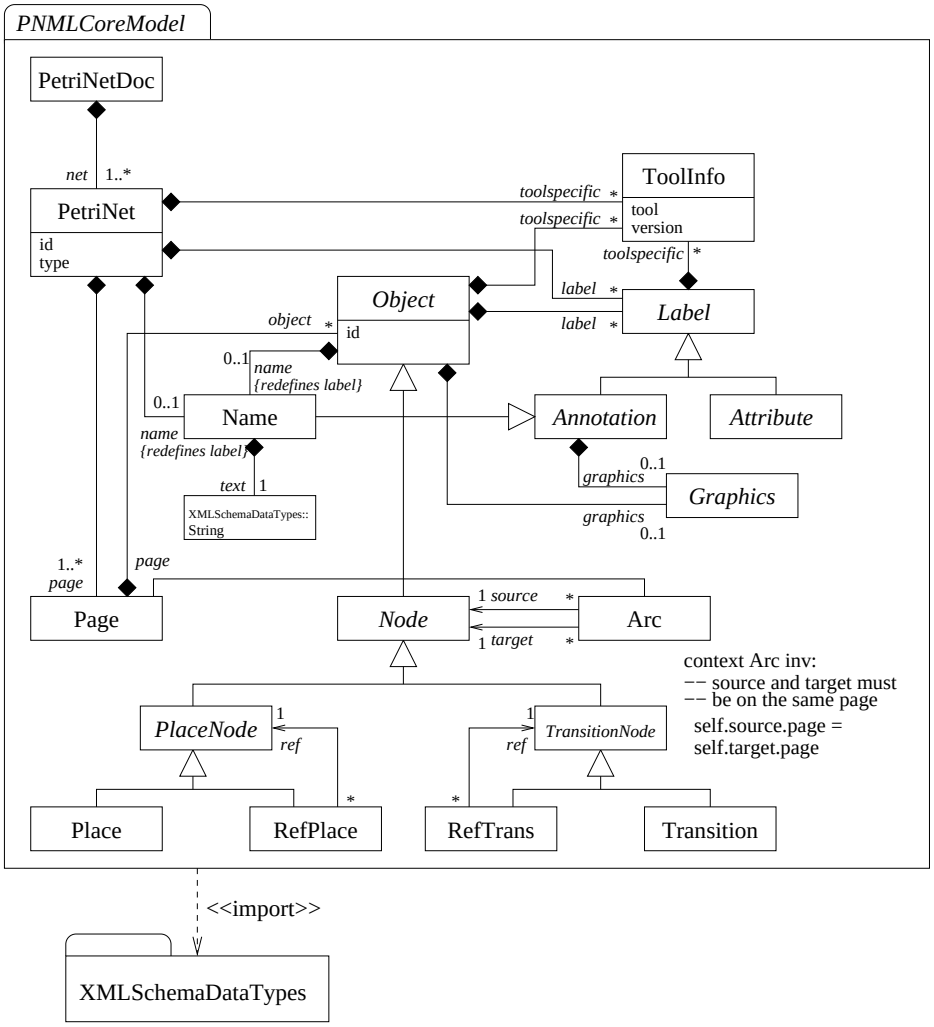

Figure 2.1: The PNML core model of ISO/IEC 15909-2:2011 [8]

labels are defined in separate Petri net type definitions, which are discussed in Sect. 2.2.2.

In addition to the concepts and relations between them, the PNML core model states also some restrictions on the structure of PNML models. For example, there is an OCL constraint stating that arcs can connect only nodes that are on the same page. Note however, that there is no constraint in the PNML core model, which states that arcs can run between a place and a transition or the other way round only. The reason for not having this

constraint in the PNML core model is that there are some kinds of Petri nets that would allow arcs between places or between transitions. This is why these kind of restrictions would be part of a Petri net type definition.

Note also that the PNML core model does not specify concrete tool specific extensions. It is up to a tool to define what it needs. But, any tool must be able to read – and later write – any tool specific extension; their contents however, can be ignored.

2.2.2 Petri net type definitions

As stated above, it is the purpose of a *Petri net type definition* to define which labels are possible in a specific kind of Petri net, and also to define some additional restrictions on the legal connections. Here we explain this idea by the help of a simple example: Place/Transition-Systems (P/T-Systems in short).

The two additional kinds of labels for Place/Transition-Systems are the initial marking for places, and the inscription for arcs. The initial marking can be any natural number (including 0) and the inscription for arcs can be any positive number. Figure 2.2 shows the UML model for these concepts and how they are related to the concepts of the PNML core model.

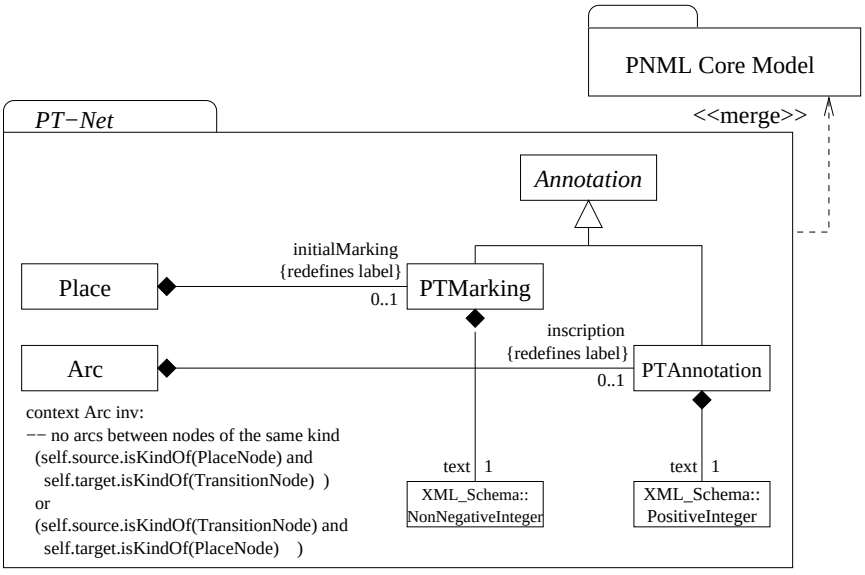

Figure 2.2: The PNTD for PT-Nets

In Fig. 2.2, there is also one additional OCL constraint. Without going

into the details of OCL, this constraint states that an arc must run from a place to a transition or from a transition to a place. So, for P/T-Systems, it is no longer possible to connect places with places or transitions with transitions.

A Petri net type definition, would typically also define how the new concepts from Fig. 2.2 would be mapped to XML. If not stated, the ePNK will uses a default of how the new features are mapped to XML. For the example above, this default mapping is good enough (and actually compatible with ISO/IEC 15909-2).

2.2.3 Mapping to XML

As mentioned above, the PNML core model together with the model for a Petri net type definition, define the concepts of a specific kind of Petri net and how they can be connected. Therefore, these models are the centerpiece of PNML. Still, PNML is an XML transfer format for Petri nets. So, PNML defines how these concepts are saved or represented in XML. This is achieved by mapping every concept or feature of the UML models to some XML construct.

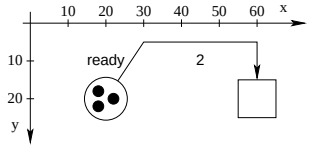

Figure 2.3: A simple P/T-System

Here, we do not give these mappings, but rather show an example (for a detailed discussion of the mappings, see [6]). Figure 2.3 shows a simple example of a P/T-System in its graphical representation (concrete syntax); Listing 2.1 shows its representation in PNML's XML-syntax[1].

Note that the listing also shows an example of a tool specific extension: the positions of the individual tokens in the place.

2.3 ePNK: Objective

The main objective of the ePNK is to fully support the concepts of PNML, so that new Petri net types along with the mapping to XML syntax can be

[1]We deleted some line-breaks to make this listing fit to single page

Listing 2.1: PNML code of the example net in Fig. 2.3

```
<pnml xmlns="http://www.pnml.org/version-2009/grammar/pnml">
 <net id="n1" type="http://www.pnml.org/version-2009/grammar/ptnet">
  <page id="top-level">
    <name><text>An example P/T-net</text></name>
    <place id="p1">
      <graphics><position x="20" y="20"/></graphics>
      <name>
        <text>ready</text>
        <graphics>
          <offset x="0" y="-10"/>
        </graphics>
      </name>
      <initialMarking>
        <text>3</text>
        <toolspecific tool="org.pnml.tool" version="1.0">
          <tokengraphics>
            <tokenposition x="-2" y="-2" />
            <tokenposition x="2" y="0" />
            <tokenposition x="-2" y="2" />
          </tokengraphics>
        </toolspecific>
      </initialMarking>
    </place>
    <transition id="t1">
      <graphics><position x="60" y="20"/></graphics>
    </transition>
    <arc id="a1" source="p1" target="t1">
      <graphics>
        <position x="30" y="5"/>
        <position x="60" y="5"/>
      </graphics>
      <inscription>
        <text>2</text>
        <graphics>
          <offset x="0" y="5"/>
        </graphics>
      </inscription>
    </arc>
  </page>
 </net>
</pnml>
```

easily plugged into this tool – and to provide all the Petri net type definitions for the types defined in ISO/IEC 15909-2:2011.

As soon as such a new Petri net type definition is plugged in, it should be possible to load and save Petri net documents that contain nets of these types. Moreover, there should be a graphical editor that allows us to edit Petri nets of any plugged in Petri net type; and the editor should be fully aware of all the features (annotations and attributes) and the additional constraints of the plugged-in Petri net types.

For tool developers, the ePNK should provide an API to easily load and access Petri nets from PNML files, to manipulate them, and to save them. Moreover, it should be easy to plug in new functionality for analysing Petri nets and visualizing the results, and for manipulating Petri nets or transforming them to other models or code.

2.4 How to read this manual

In this manual, we will explain the features of the ePNK in more detail.

On the one-hand side, this manual covers the parts relevant for the "end user" who just wants to load, save and edit Petri nets of existing types and use some existing or plugged in functionality of the ePNK. In the rest of this manual, we call these "end users" just *users*. All the information relevant for users of the ePNK can be found in Chapter 3.

On the other-hand side, this manual covers the information relevant for *developers* who are interested in using the ePNK for their purposes: extending it by defining new Petri net types, by defining new tool specific extensions, or by implementing new functionality. Chapter 4 provides the information relevant for developers who want to extend the ePNK.

In some future version of this manual, there will also be a part that discusses the architecture, the design, and some of the used technologies (and their problems). These parts might be interesting for developers, but actually addresses people interested in model-based software development technologies that are used (and extended) in this project: EMF, GMF, Xtext, ExtendedMetaData, EMF Validation, and OCL. For now, we conclude this manual with Chapter 5, which gives a brief experience report on the implementation of the ePNK and an overview of features of the ePNK that might be implemented in the future.

Chapter 3

Users' guide

This chapter explains how to use the ePNK for creating, loading, saving, and editing Petri nets, and also how to use some of its functions. Since new Petri net types can be plugged in, we try to point out the general principles of these editors and how to use them. For the particular syntax of some labels of a specific Petri net type, it might be necessary to refer to the documentation of the specific Petri net type. We will discuss these principles by some of the Petri net types that come with the basic version of the ePNK; and we use high-level nets (in terms of the ISO/IEC 15909-2 *High-level Petri Net Graphs*, or *HLPNGs* for short) to point out for which parts you would need to refer to the specific documentation of the specific Petri net type.

3.1 Eclipse as an IDE

For users who are new to Eclipse and its IDE (Integrated Development Environment), we start with a brief overview of Eclipse's workbench. Users who are familiar with Eclipse already can directly read on in Sect. 3.2.

Once you installed and started Eclipse (see Chapter 1), you see the Eclipse *workbench*. Depending on the chosen *perspective*, the different parts can be arranged in different ways. But, the principle behind is always the same. Figure 3.1 shows an example of the Eclipse workbench, with some numbers marking some parts, which we will discuss next.

At the top of Fig. 3.1 marked by (1), you can see the *menu bar* and the *toolbar*. Here, you will find the menus and tools for all the standard functionality, such as loading and saving files, and for standard editing operations. The menus that are shown in the menu bar depend on your installation and

also on the editor that is currently active. For many operations, there are also the standard shortcuts, like CNTRL-S (on the Windows platform) for saving the contents of an editor to a file. For getting more information on that, you could chose the "Help Contents" in the menu "Help" in the menu bar, and read the "Workbench User Guide".

> **Note:** In automatically generated editors, such as the graphical editor of the ePNK, the copy/paste functionality with CNTRL-C, CNTRL-V, and CNTRL-X does not work properly. In order not to mess up models, by improperly using cut/paste operations, the graphical editor of the ePNK does not support copy-/paste yet.

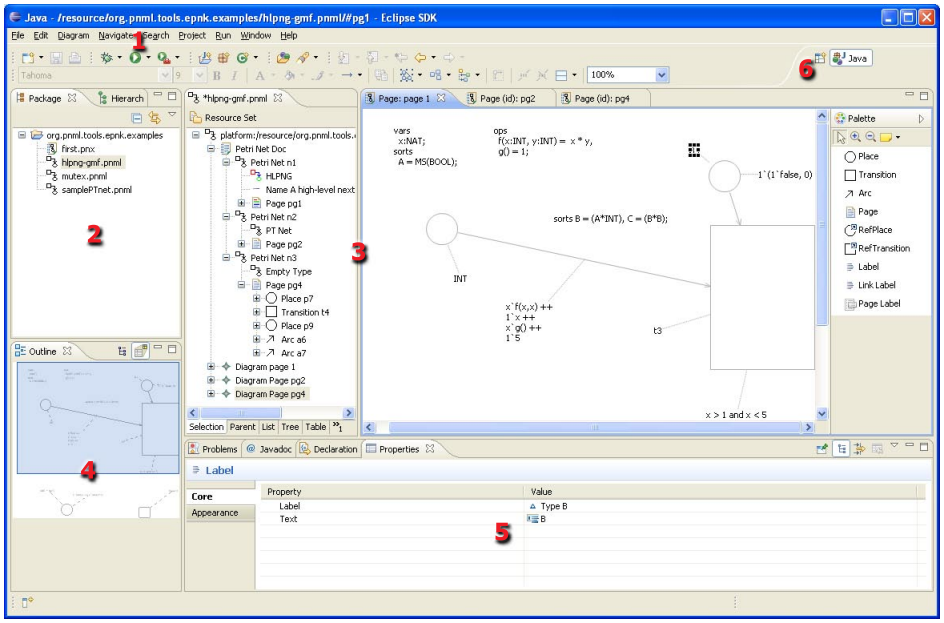

Figure 3.1: The Eclipse workbench

On the left-hands side, marked by (2), you can see the *package explorer*, which gives you access to all the files in your workbench. The package explorer can be used for browsing through the existing files and for manipulating, renaming, copying, moving, and deleting them. This is very much like the file explorer of your operating system. Eclipse actually has different kinds of explorers, depending on the perspective and the user's preferences.

The package explorer is made for Java development projects. For our purposes, any of these explorers would do, like for example the "navigator" or the simple "project explorer". To find and open one of these other explorers, you can use the menu "Show View" in the menu bar menu "Window". All these explorers have some important concept in common, which concerns the organisation of files in the workbench: the top-level "folders" are actually not folders, but they are *projects*. This is relevant only when creating these projects. You can create a folder or file in the workbench only after you have created some project; this can be done via the "File" menu or by a right-click in the explorer and then selecting "New"→"Project". Note that in the dialog, you can create many different kinds of projects; for us the kind "Project" in category "General" will do. Then files can be created within this project.

In the center (3), you can see the *editor area* of Eclipse. This is where all the editors that are started in Eclipse will be opened. Note that there can be many editors open at the same time (in our example, there are four editors open). Typically, you can only see one at a time and the others are hidden below it. But, you can move the editor tab to some border of the editor area, so that you can see the contents of two or more editors at the same time. In our example, there is a tree editor of a complete Petri net document open on the left-hand side, and, on the right-hand side, you can see one specific page open in a graphical editor (the graphical editors for some other pages are hidden beneath). Note that, even though there can be many editors open and even visible at the same time, there will always be only one editor that is active. This editor and what is selected in it determines what you see in some other views. For example, you can see the *outline view* (4) of the page or you can see the property of the currently selected element in the *properties view* (5) at the bottom. In order to open an editor on some resource in the explorer, you would, typically, double click on the resource you want to open. This will open the *default editor* on the selected resource. You can also use the right mouse button on a resource to open a pop-up menu and then select "Open with" to select a specific editor for this purpose. The way of editing the contents of a resource depends on the kind of editor (mostly, it is straightforward). Saving the file can typically done by a shortcut (like CNTRL-S) in all editors (or via the "File" menu in the menu bar). An editor can be terminated (closed) either by clicking the close symbol on the tab of the editor or via the "File" menu in the menu bar.

Most editors support undo and redo of the latest changes, which you can access via the "Edit" menu or via the CNTRL-Z and CNTRL-Y shortcuts.

Note that the graphical editor of the ePNK cannot be initiated directly from the explorer since the resource could have many pages and a page is not the top-level element. When you open a PNML file, a tree editor will be opened that shows the structure of the Petri net. The graphical editors for a page of a Petri net can be opened by a pop-up menu (right mouse button) on pages in the tree editor for Petri nets or by a double click on the page (see Sect. 3.4 for details).

All the other areas of the workbench are *views*[1]. In Eclipse, views are used for many different purposes. The views that are most relevant for us, are the *outline* (4), the *properties* (5), and the *problems view* (not visible in Fig. 3.1). The outline gives an overview of the contents of the currently active editor and in case of a graphical editor allows us to quickly move around the visible area of this editor. The properties view shows some details of the currently selected element in the editor; in many cases, the properties view also allows us to edit some properties. Note that, initially, the properties view might not be open. You can, typically, open it from the active editor via a context menu on the right mouse button: In the pop-up menu that opens, there will be a menu "Show Properties View", which opens the properties view. You can also open the properties view via "Window"→"Show View"→"Others ..." and then selecting "Properties" in the category "General".

We mentioned already that the Eclipse workbench can appear in different ways, which is defined by the chosen *perspective* (and some user-specific settings). The perspective can be changed via the tools at the top-right of the workbench, which are marked with (6) in our example. We do not need to change it; but if, for whatever reason, you end up in a wrong perspective, by clicking on the left symbol, you can open the "Open Perspective" dialog. There, you can select the perspective "Resource" or, if you like, "Java" (which is the default perspective).

If you are interested in more details in the Eclipse workbench, you can have a look into the Eclipse help ("Help"→"Help Contents") or at one of the many books or online articles; `http://www.vogella.de/articles/Eclipse/article.html` could be a start.

3.2 Creating Petri net files

This section explains how to create new ePNK files. Note that there are two formats in which the ePNK can save a Petri net. The first and recommended

[1] Actually, also the resource browsers are views.

format is PNML. The second format is the XMI-serialisation of the PNML models, which we call PNX. Note that PNX, is part of the ePNK since XMI is the standard serialisation mechanism of the technology (EMF and Ecore), and came for free. Whether PNX really should be a part of the ePNK distribution is yet to be seen. Therefore, the focus of this users' guide is on PNML.

The easiest way of getting started with the ePNK is obtaining existing PNML files from somewhere else and just copy them to the workbench. For example, you could get some examples from the ePNK home page: `http://www2.imm.dtu.dk/~ekki/projects/ePNK/`. You can also use a text editor and create a simple text file with file extensions ".pnml" and insert the single line

```
<pnml xmlns="http://www.pnml.org/version-2009/grammar/pnml"/>
```

to this file, which is an empty Petri net document without any nets in it.

The ePNK also provides you with a wizard for creating a PNML document. Like all Eclipse creation wizards, this wizard will be started via the "New" menu, which can be either accessed by the "File" menu from the menu bar or via the pop-up menu that opens on a click to the a right mouse button in the explorer. Then, select "Other..." (the short-cut to that would be pressing CNTRL-N in the explorer) and in the newly opened "Select a wizard" dialog choose "PNML Document" from the "ePNK" category and press "Next". In the next dialog, you must choose a name and, if you want, you can choose a different folder in which this file should be created. Pressing "Finish" will create the file; then, the newly created file will be opened in a tree editor (see Sect. 3.3); note that you also can continue the creation process by pressing "Next", which will allow you to chose an XML-Encoding. Note that, in the dialog with the encoding, there is also a field asking for the "Model Object"; but you cannot choose anything here since PNML, in contrast to other formats, has a fixed root object that cannot be changed: "PetriNetDoc".

Note that in the same wizard category "ePNK" there is another wizard called "PNX Document". When you use this wizard, a PNX file will be created. In this wizard, you can select a root element different from the PetriNetDoc – but this would be reasonable only in very special cases (and when you know exactly what you are doing).

3.3 The tree editor

As mentioned earlier, the ePNK provides two kinds of editors for Petri nets. The *tree editor*, which allows us to create, modify, and delete all parts of the Petri net in a tree like structure. And there is a *graphical editor* in which a page of a Petri net with its places, transitions, and arcs can be edited in a graphical way. Clearly, the graphical editor is more convenient for editing pages than the tree editor. But, other parts like for example the page structure and the complete Petri net document are more convenient to edit in the tree editor. This is why there are two different editors in the ePNK. The graphical editor for pages is always started from a selected page – either in the tree editor or in the graphical editor. When opening a PNML document from the resource explorer, it will always be the tree editor that opens. A graphical editor can be opened by a double click on a page element in an already open editor.

3.3.1 The tree editor: Overview

Let us have a closer look at the tree editors first. Figure 3.2 shows the Eclipse workbench with two PNML documents open in tree editors. The right one shows the tree editor opened with the PNML file ("test.pnml") with the single line as discussed in Sect. 3.2. Therefore, it contains only the Petri net document element without any contents. The other PNML document, which is open on the left-hand side ("hlpng-gmf.pnml"), shows a Petri net document with three nets that have different types.

These documents were opened from the explorer by a double click[2] on the respective file in the workbench's explorer. Let us briefly go through what you see in the Petri net document "hlpng-gmf.pnml". The top-line shows the actual *resource* or file in which this document is stored; the second line is the symbol for the Petri net document itself – all documents will follow this structure in the tree editor. Then you can see that there are three Petri nets contained in this document (with *ids* n1, n2, and n3); the last net is actually not folded out because its was not fitting to the screen. The first line below the Petri net is the type of the Petri net. The first net is a high-level net, which is named *HLPNG* according to ISO/IEC 15909-2; the second net is a Place/Transition-System, called PTNet according to the

[2]Remember, that you can use the pop-up menu to make an explicit choice by which editor you want to open the file. This way, you can open the file with a text editor, so that you can see the PNML it produces. On a double click, the file is opened with the editor you had selected the last time.

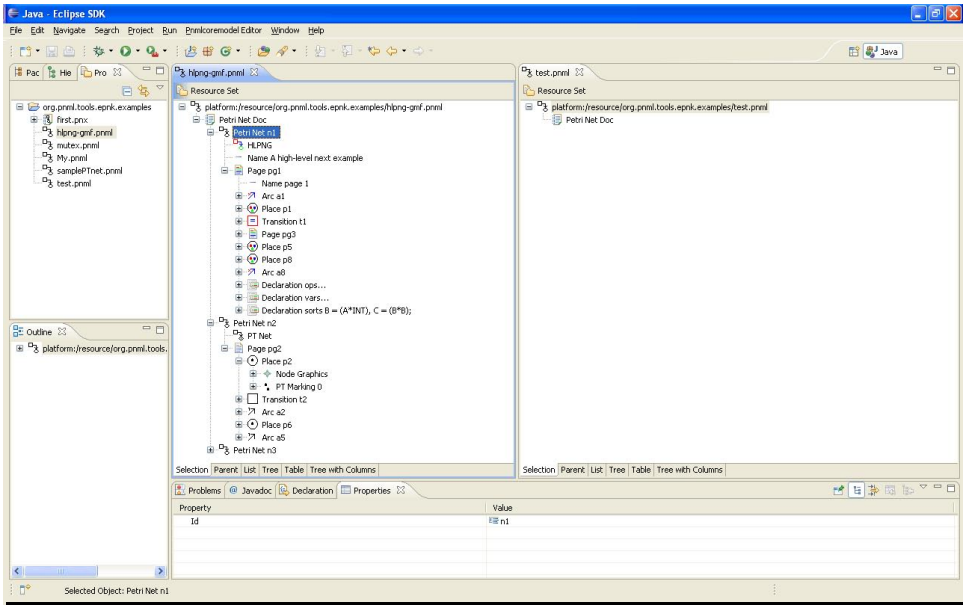

Figure 3.2: Two Petri net documents in tree editors

standard. You can also see that the nets contain places, transitions, and arcs, which are indicated by corresponding icons. You can also see some pages and sub-pages. Note that the icons for the places, transitions, and arcs are different for the two different Petri net types, so that it easier to distinguish them on first glance. In the properties view at the bottom, you can see the properties of the currently selected element, which is Petri net n1, and the only property is its id. Note that this net also has a name; this, however, is not shown as a property, but as a *child element* of the net, which is true for all labels of Petri nets. In this case, the name is "A high-level next example".

3.3.2 Creating elements

You can unfold all the sub nodes (children) of the net and this way inspect the complete document in all details. More importantly, however, you can create the basic elements of the Petri net document. You can create new nets (along with their type) and their pages. And from there, you would use the graphical editor to draw the rest. This basically works by inserting *child elements*. Inserting a child element is done by right-clicking on the element to which you want to add a child, then selecting "New Child" in the dialog

that pops up, and then selecting the appropriate element. Figure 3.3 shows the pop-up dialog when inserting a new Petri net to the Petri net document.

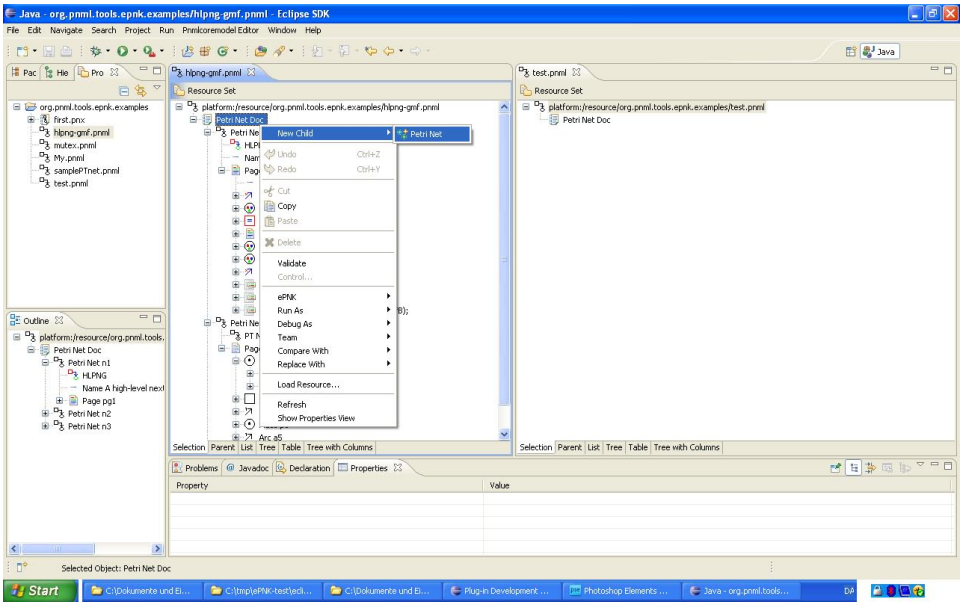

Figure 3.3: Pop-up menu when inserting a new Petri net

Note that this dialog will show all available Petri net types, from which you will need to select. Then a new net of that type will be created. Note that the created net will contain a child element, which represents the type of this net. You should never delete or change this net type manually[3]. You will find some more information on the Petri net types that are deployed together with the ePNK in Sect. 3.5.

After you have created a new Petri net, the name of the net and new pages can be inserted to it by via the "New Child" pop-up menu on the selected Petri net similar to creating the net – just select the kind of element you want to create.

3.3.3 Saving the document

As discussed in Sect. 3.1, you can save the net via the "File" menu or with the CNTRL-S shortcut. Note that saving the net must be done – and can be done only – in the tree-editor. Therefore, only the tree-editor shows the dirty-flag, when the editor contains unsaved changes.

[3]Unless you know exactly what you are doing.

3.3.4 Validating and correcting the document

Before saving a PNML document, it would be a good idea to *validate* the net. This will check whether all the constraints that PNML imposes on Petri nets are met. It is possible to save a document that does not properly validate, and you would be able to load the file again. But, if you save a file that does not properly validate, you cannot be sure that the saved document is ISO/IEC 15909-2 conformant PNML, and other tools might not be able to load it.

There are many things that can be wrong and need validation on a Petri net document. Most of them are type specific (such as the requirement that an arc must run from places to transitions or the other way round only); these Petri net type specific constraints will be discussed in Sect. 3.5. But, there are also some general constraints:

- Every Petri net object must have an id and this id must be unique in the scope of this document.
- Arcs may only connect nodes which are on the same page (as long as you are using the graphical editor, this constraint cannot be violated; but if you do changes in the tree editor, this could be violated).
- There must not be cycles on the references between reference nodes, and all reference nodes must refer to a node.
- A reference node must refer to a node within the same net.

In order to identify which constraints are violated, you can use the validation feature. Click on the right mouse button on the Petri net document; then, in the pop-up menu, select "Validate". The result of the validation will be shown in a dialog; the results of the validation is also visible in the *problems view*[4] after the validation as shown in Fig. 3.4. Most of these errors are actually coming from high-level nets. But, there is also a general constraint violated in this example: some IDs collide (line 5 and 6 in the problems view), which means that the same ID is used twice in this document.

Note that you can double-click on the individual problems in the problems view. Then, an editor is opened with the element to which this problem refers to selected. If there is a graphical editor open for the page that contains the element with the error, the ePNK will show the selected element in the graphical editor; if there is not graphical editor open with that element, the element will be shown in the tree editor.

[4] If the problems view is not open, you can open it by "Window"→"Show View" →"Problems".

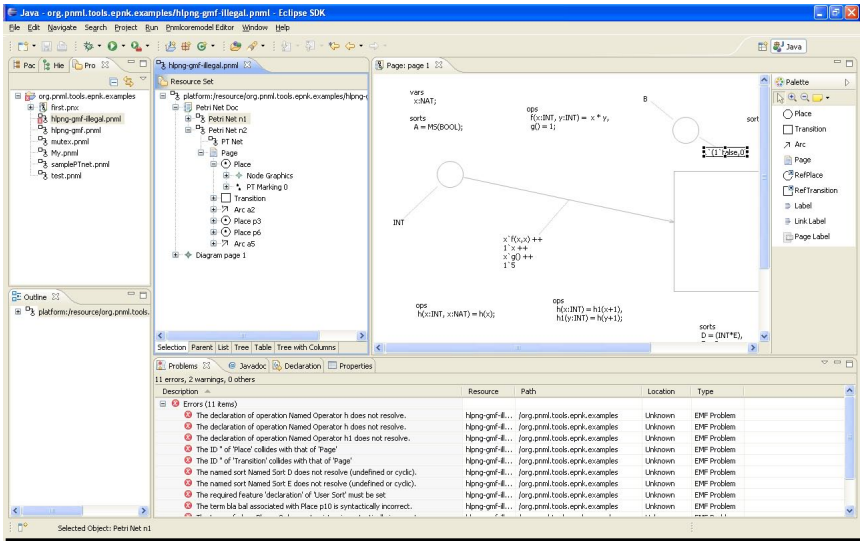

Figure 3.4: The problems view with many constraint violations

In order to reduce the number of errors, you can also do a validation on sub-elements of the Petri net document, which could be a net or even a single page. Ultimately, however, you must validate the complete Petri net document.

In our example, there are some problems that can be fixed automatically. For example, ids can be set automatically. The ePNK provides an action for this. To this end, select the Petri net document, click the right mouse-button, and then select "ePNK"→"Add missing IDs" in the pop-up menu. This will fix all problems with the ids within a Petri net document. Actually, there is a shortcut: a double-click on the Petri net document element in the tree editor will automatically add all the missing ids.

Some other errors might need some manual changes, which could typically be made in the graphical editor of pages.

3.3.5 Other Petri net information

In principle, you can inspect and edit all the information of a Petri net document in the tree editor. In our example from Fig. 3.3, you can also see some labels (declarations of high-level nets in this case, or a marking of a P/T-System) or graphical information. If you have a closer look at these examples, you will also find some other types of elements such as tool specific information – and once graphical editors are started some auxiliary

data. But, it is strongly recommended not to change any of this information in the tree editor[5].

3.4 The graphical editor

For editing the contents of pages, the graphical editor should be used. The graphical editor can be opened by right-clicking on the respective page in the tree-editor and then, in the pop-up menu, selecting "ePNK"→"Start GMF Editor on Page". A shortcut for this is double-clicking on the page.

When you open a graphical editor on a page for the first time, you will be warned that this change cannot be undone – and no undos will be possible beyond that point. Therefore, you will be asked whether to proceed with that operation or not.

Figure 3.4 shows the graphical editor with a page open in a graphical editor; this is a page from a high-level Petri net (in this case one with several errors in it). Normally, this new editor shows on top of the tree editor, but it can be moved to the right side (click in the tab at the top of the editor window and move it while keeping the mouse pressed), so that the tree editor and the graphical editor are visible at the same time.

Figure 3.7 shows another example of a Petri net opened in the graphical editor, which we will discuss in Sect. 3.4.3.

3.4.1 Overview of the graphical editor

On the left-hand side of the graphical editor for the page, you see the *canvas* with all the Petri net objects on that page represented in a graphical way. This includes also the *labels*, which are either attached to an object by a dashed line or attached to the page itself, in which case it is called a *page label*.

At the top, you see the *tab* of this page, which shows the page's name (if the page has a name label assigned to it) or its id, or the path to this page (if the page has neither a name nor an id).

On the right-hand side, you see the *palette* or tool bar of the graphical editor. These tools allow you to create all the Petri net objects. Note that you can also create sub pages.

[5]Once the features of the ePNK that are really needed in the editor are fixed, the parts that should not be edited in the tree editor will probably removed or at least made read only.

There are two different tools for labels. The tool "Label" is for creating labels that are attached to objects, the tool "Page label" is for creating labels that are directly attached to the page that is shown in this editor.

For creating objects and labels, you first select the tool by clicking on it, and then clicking somewhere into the canvas. For creating an arc, you select the arc tool and then click on the source object, and keeping the mouse pressed and move the mouse to the target object. Note that the arc is not added between two objects, if the Petri net type you are editing does not allow this.

3.4.2 Labels

When creating a new page label on a page, the graphical editor will show you all the possible options of legal labels for that type of net Petri net via a pop-up menu. Figure 3.5 shows the pop-up menu during the creation of a page label for a high-level net, where the only option "Declaration" is shown here. You can select an option, after which a label of that kind will be created. You can also abort by either pressing the "ESC" button or clicking somewhere outside the menu.

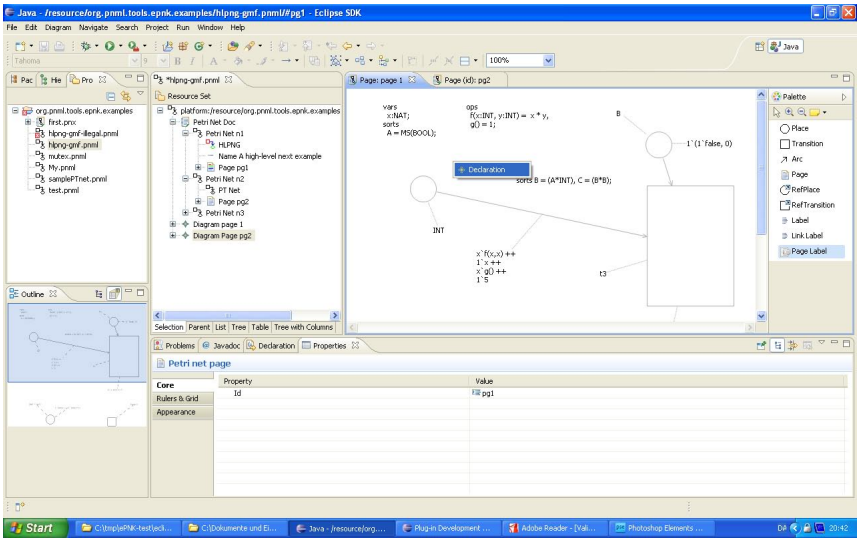

Figure 3.5: Pop-up menu during page label creation

The process for creating a label and attaching it to an object is slightly different. First you must create the label. This new label, however, will not be attached to any object yet, which is indicated by the text "<not connected

label>". A not yet connected label can – and must – be connected to some Petri net object (which could also be a sub-page) by choosing the tool "Link Label", clicking on a label, and then without releasing the mouse button moving it over the object the label should be attached to. Then, a pop-up menu will be opened showing you the possible kinds of labels that could still be attached to the chosen object. This is shown in Figure 3.6 for a label that is attached to a place of a P/T-System. The possible options are "Name" (which is a legal option for any object, but only if there is no name attached yet) or "PTMarking 0", which is the initial marking for P/T-Systems (where "0" is the default value). After the selection, the label of the chosen type will be attached to the object. Again, attaching the label can be aborted by pressing "ESC" or by clicking somewhere outside the pop-up menu.

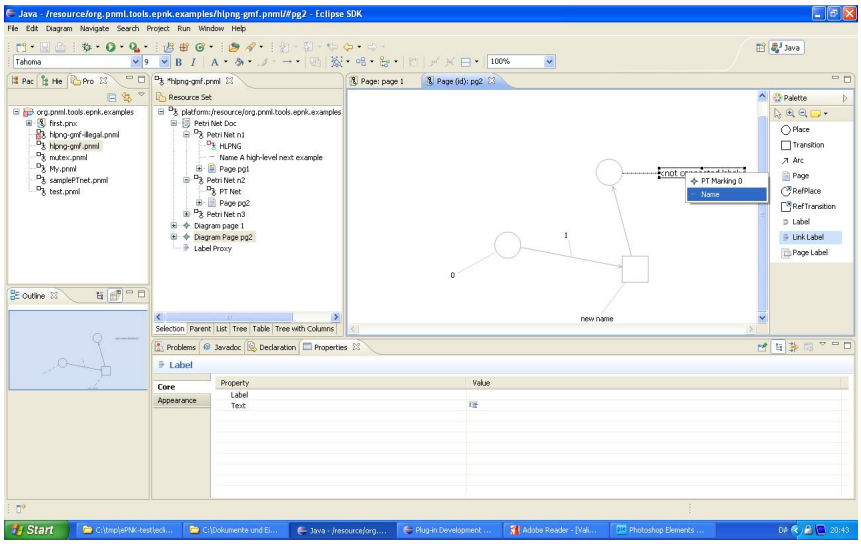

Figure 3.6: Pop-up menu during attaching a unconnected label to an object

After the label has been attached to an object, it can be edited "in place", by clicking into it and pressing the ENTER key in the end. The legal syntax of the label depends on the Petri net type and which kind of label it is. In general, when editing labels and page labels there are two different cases: The first case are *simple labels*, which typically are simple values like "true" or "false", values like numbers, arbitrary strings, or IDs (in general, it will be some form of data types). If such a label is typed in syntactically incorrect, the new value will be rejected, and the value of that label will be reverted to the value it had before editing. The other case, are

structured labels. These are, typically, labels with a complex syntax, as for example the declarations of a high-level net (actually all labels of high-level nets except for the names are structured). All these labels will be parsed and checked for syntactical correctness; but the entered text will be stored in all cases. If the text is syntactically incorrect, however, the structure is not set and this will very likely result in some validation error later (see Sect. 3.3.4). So this error needs to be fixed, by editing the label again. In case of such an error, the label will be marked with a warning symbol (and when the mouse is moved over the warning symbol, the tool tip will indicate that the label could not be parsed). If, for example, we delete the comma that separates the two sort declarations in the label "sorts B = (A*INT), C = (B*B);" the label will be decorated with a warning symbol. Upon validation, a validation error message will be given and later shown in the problems view.

The documentation of the legal syntax of these type specific labels, in particular the one of the structural labels, is part of the documentation of the Petri net type definition. For the types deployed together with the ePNK, this information can be found in Sect. 3.5.

Note that labels, in principle[6], can have line-breaks. Since pressing the ENTER button will finish the editing of a label, however, a line-break is inserted to a label by pressing CNTRL-ENTER while editing the label.

3.4.3 Attributes

As discussed earlier, some "labels" of a Petri net object are not supposed to be represented as graphical annotations of a Petri net object. These are called *attributes*. Figure 3.7 shows an example of a Petri net which uses attributes for some objects. It is a signal-event net (SE-net) [26], which will be used later in the developers' guide of this manual as an example of how to define new Petri net types for the ePNK[7].

In this example, arcs have an arc type. If the arc type is not set, it is considered to be a normal arc. The arc that is selected in Fig. 3.7, is of type "inhibit", i. e. it represents an inhibitor arc. The value of the attribute is visible in the properties view, and can also be changed there. In this example, the arc is graphically shown as a lollipop – which comes from the implementation of that Petri net type, which implements a dedicated graphical representation for these kind of arcs. Another example is the signal

[6] That is, if the legal syntax of a specific Petri net type does allow it.

[7] The reason we need to resort to SE-nets here is that all the Petri net types that are defined in ISO/IEC 15909-2 use annotations only

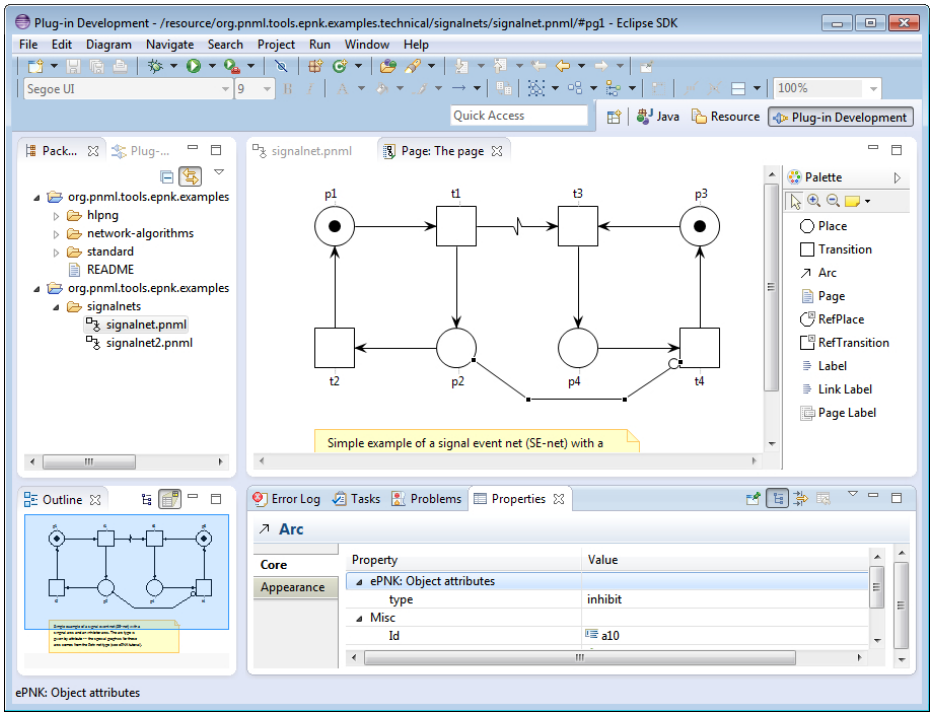

Figure 3.7: A Signal/Event net open in the graphical editor

arc (the one with the "flash" decoration), which is an arc of type "signal". For this net type, also the marking is an attribute; therefore, it is not shown as a label. It can only be changed by selecting the respective place, and then changing the attribute in the properties view. In this net type, also the marking is shown with a special graphics for the place (as black tokens).

As mentioned before, the value of an attribute can be changed by selecting the respective object and then changing the value in the properties view. Depending on the type of the attribute, this can be done by either typing some text into the value column for that attribute or by a drop down menu (if there are only finitely many possible values). If you want to reset a value to the default value (the default is that the value is not set at all), you can right-click into the property column of that property in the properties view and then select "Restore default value".

3.4.4 Pages

The graphical editor also allows you to create other pages on the page it is showing. In order to avoid confusion, we call such a page a *sub-page*. Creating a sub-page can be done with the "Page" tool, in the very same way in which places or transitions are created on a page. In the graphical editor, a sub-page is graphically represented as a rounded rectangle.

It is possible to open a graphical editor on a sub-page from the graphical editor via a pop-up menu on the right mouse button: "ePNK" → "Start GMF Editor on Page" (as we have seen it for the tree-editor). And also here, a double-click on the page in the graphical editor is a shortcut for opening a graphical editor on this page. Therefore, the tree-editor is needed only for creating the top-level pages of the net; all the sub-pages could be created by the graphical editor. But navigation to sub pages might be a bit easier and much faster in the tree-editor; this is why you would probably want to use the tree- editor for navigating and opening sub-pages further down in the tree-hierarchy. It is recommended not to create sub-pages in the tree editor, since they would not have a position in the graphical editor. Still, it is possible and the graphical editor would show these pages (as well as other objects created in the tree-editor) in the top-left corner, when it is opened with the graphical editor for the first time. Then, you could move it to a better position.

Actually, the graphical editor will indicate by a special decoration when a sub-page is open in some graphical editor: a symbol of an open folder.

What is more important about pages is how to deal with their labels. Typically, all the type-specific labels are represented as page labels on the respective sub-page. For HLPNGs, for example, the declarations of a page are shown as page labels on that sub-page. The name, however, will be shown as a label attached to that page on the super-page. But, which labels are shown as labels attached to the sub-page on the super page, and which labels are shown as page label on the opened sub-page is up to the Petri net type definition.

3.4.5 Graphical features

The graphical editor of the ePNK allows you to make all kinds of changes to the graphical appearance of the the Petri net. The features supported are the standard features of GMF editors. Figure 3.8 shows the same net as above again; in order to high-light some GMF features now with the grid and rulers switched on.

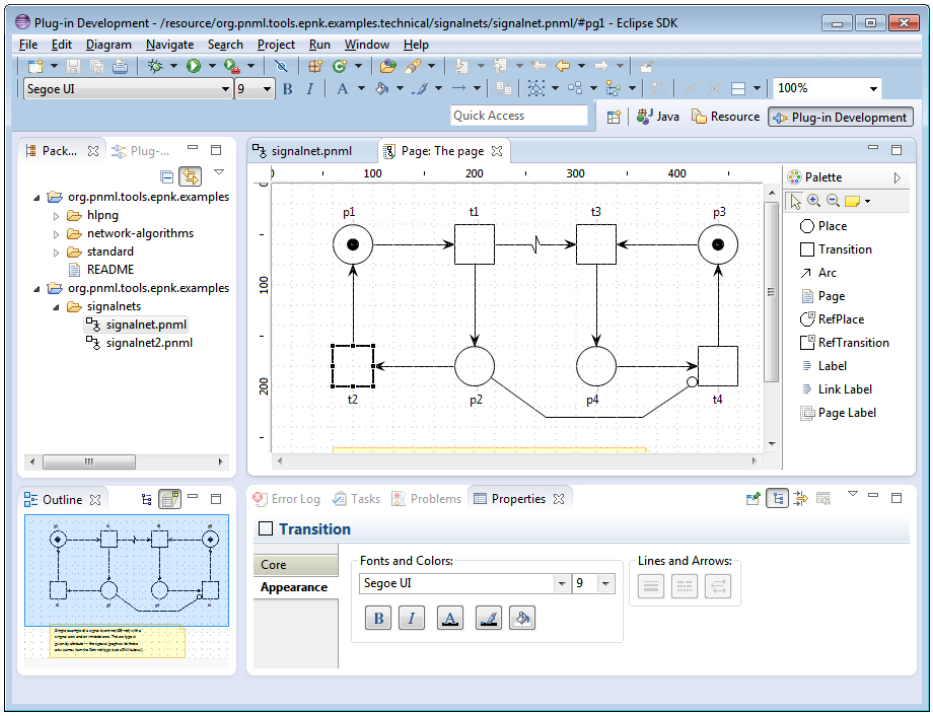

Figure 3.8: The Signal/Event net again: now with grid and rulers

Since the graphical features are pretty straightforward, and are similar to typical graphical editors, we do not explain the details here. The graphical features can be changed in the properties view, when selecting the "Appearance" section (up to now, the properties view had always shown the "Core" section). Figure 3.8 shows the graphical features that can be changed, when a node is selected. The available features vary for the different kinds of objects.

Note that Petri net types can define a special graphical appearance for some nodes or arcs, which depends on attributes or some other information of the resp. element. In that case, some graphical information selected by the user might be overruled by the specific graphical information for that element of the Petri net type. You can see an example of such a specific graphics in Fig. 3.8, where signal arcs and inhibitor arcs are shown in their usual graphical notation. In this example, however, the specific graphical information does not interfere with the user graphics. But, if a net type defines that some specific arcs should be displayed in red colour, for example, this would override the colour chosen by the user.

The ePNK saves the nets in exactly the graphical representation you see them before you save the file. But, the exact graphical representation is saved as a tool specific extension of the ePNK! This means that the exact graphical representation can be reproduced only on the ePNK.

The ePNK also transfers most of the graphical information to the respective features of PNML, so that other tools could reproduce almost the same graphical appearance as you see it in the ePNK. Some features, however, are not supported by PNML (as for example the size of labels) and some others are not yet transferred by the ePNK to PNML. In turn, some graphical features of PNML are not supported by the ePNK editor (e. g. the alignment of text in labels).

In Sect. 3.7.3, you will find a complete list of graphical information that is transferred to PNML elements. Here is a brief list of graphical information from the graphical ePNK editor that is *not* transferred to PNML: font styles (bold or italic for labels); routing and jump link information for arcs. Note that also the comments that you can place on a page will be lost in other tools, since this is tool specific information only in the ePNK (comments are not a concept of PNML).

Note that PNML nodes that do not have a size attached to them, will have the width and height 40pt (the GMF default).

Arcs can have *intermediate points* or *bend points* in the ePNK as well as in the PNML. These intermediate points will be transferred to the PNML model. But there is some caveat: when a node is moved in the ePNK editor, the bend points of the attached arcs might also be moved. Due to some quirk in GMF, these changes are not transferred to the PNML model at all. If you want to be sure that the intermediate points of the PNML model corresponds to what you see in the graphical editor of the ePNK, you should move at least one bend point of each arc attached to the node after you have moved the node.

Arcs are either drawn as a *polyline* or as a *bezier curves*, which is defined by the "Smoothness" chosen for the respective arc in the ePNK editor. The ePNK draws an arc as a polyline, if the "Smoothness" is set to "None"; for all other choices of "Smoothness" (i. e. "Normal", "Less", or "More") the arc will be drawn as a quadratic bezier curve, where every second intermediate point is used as a control point as mandated by ISO/IEC 15909-2. The information on whether an arc should be drawn as a straight line or as a bezier curve is transferred to the PNML model.

The graphical editor of the ePNK also supports the image feature of PNML: Nodes can be assigned an image in the PNML model; instead of the normal shape of the respective node, the image will be shown. The ePNK

supports JPEG and PNG – as required by PNML. Actually, the ePNK supports even more graphics formats[8]; for example GIF is supported. But, since these formats are not supported by PNML, we recommend to use the JPEG and PNG format only.

Right now, the image attribute of a node (in the Fill element of the NodeGraphics) can be set in the tree editor only. From version 1.0.1 on, the properties view will have an image property, in which the path to the image can be set directly when the resp. element is selected in the graphical editor.

Note that, for efficiency reasons, each image is loaded only once – the first time it needs to be shown in the graphical editor. If you change an image file and you want a graphical editor in an already open document to show the new image, the image cache of the open editor must be cleared explicitly. To this end, right-click on the top-level "Petri Net Doc" element in the ePNK tree editor of that document and select "ePNK→"Clear Image Cache".

3.5 Petri net types

In this section, we give an overview of the Petri net types that are deployed together with the ePNK. Currently, these are P/T-Systems (PTNet) and high-level Petri nets (HLPNG). And there is the empty type (Empty), which, however, does not contain any concepts in addition to the PNML core model; therefore, we do not need to discuss this here. The empty type was introduced to explicitly indicate, that there are no Petri net type specific extensions. If you have the ePNK tutorial installed, there will be some more examples of Petri net types, which will be used in the developers' guide to explain how to implement new Petri net types.

Actually, HLPNGs come in different levels or kinds: "dot nets", which are a way of representing P/T-Nets as high-level nets; basically, "dot nets" are high-level nets restricted to the sort "DOT" and a minimal version of operators on them; "symmetric nets" are a restricted version of high-level nets that uses some special finite sorts and a limited set of operations only; and the full version of high-level nets. The kind of a HLPNG can be changed by selecting the HLPNG type in the tree editor and by selecting the kind attribute in the properties view (identified by the respective URI as defined by ISO/IEC 15909-2). For a detailed discussion of the legal constructs of the different kinds of HLPNG, we refer to [6]. Note that, in contrast to the

[8] All graphics formats supported by the Eclipse SWT `ImageLoader` should work.

Petri net type, the kind of a HLPNG can be manually changed anytime, since the kind of HLPNG concerns the validation only. The PNML syntax is the same.

3.5.1 PTNet

We start with explaining the details of PTNets. In Sect. 2.2.2 we have already seen the additional features of PTNets, which are the initial marking for places and the inscription for arcs. Both labels are simple labels, which means that it will be checked right after editing a label whether the label is syntactically correct (see Sect. 3.4.2); if it is not correct, the value will be reverted to the value it had before.

The marking of a place must be a non-negative integer in any reasonable representation[9]. The arc-inscription is similar, just that it must represent a positive integer (i. e. must be a number greater than 0).

Moreover, PTNets have the restriction that arcs may only run from a place to a transition or from a transition to a place, which will be enforced in the graphical editor[10]. Actually, the constraint is slightly more complicated due to reference nodes: We can connect place-like nodes (PlaceNodes) with transition-like nodes (TransitionNodes) and vice versa; but semantically, i. e. when flattening reference nodes, this amounts to the above condition.

3.5.2 HLPNG

HLPNGs are much more involved than P/T-Nets and we cannot explain them in all details here. For a detailed motivation and full account on what *HLPNGs* are, we refer to ISO/IEC 15909-2 [8] or [6]. Actually, HLPNG are conceptually quite close to coloured Petri nets [9] or algebraic system nets [18].

For HLPNGs, there are the following labels (in addition to names):

Declaration A *declaration* is a page label, which is used to define *variables*, *sorts*, and *operators*, which can then be used in the other labels. Every page can have any number of declarations and, within a single declaration, different kinds of declarations can be mixed. Note that all declarations are global (known in the complete Petri net), even though they are attached to a specific page.

[9] For those who want to bother with the technical details, it is any String that would be accepted by the Java `Integer.parseInt()` method as a number and evaluates to a number greater or equal than 0.

[10] In the tree editor, illegal arcs could be created, but this would not pass validation.

Declarations do not need to be contained in a page at all – they can be contained directly in the net. We do not recommend to make use of that; but ISO/IEC 15909-2 mandates this to be possible. So, the ePNK can read nets with declarations which are directly contained in the net, now[11].

Type A *type* is a label that is associated with a place. Every place must have exactly one type label which denotes the sort (which can be built from built-in sorts and sort constructs or from user defined sorts) of the tokens on that place.

HLMarking A *marking* is a label that is attached to a place and indicates the place's initial marking. The marking is represented by a ground-term[12], which must be a multiset over the place's type. Note that this label may be omitted, in which case the initial marking is considered to be empty. There can be at most one label of this kind.

Condition A *condition* is a label that can be attached to a transition. The condition is a term of type boolean and can contain variables. There can be at most one condition; if the condition is missing, it is assumed to be true.

HLAnnotation An *arc annotation* is also a term that may contain variables. The term must be a multiset term over the type of the place to which the arc is attached. Every arc should have exactly one arc annotation[13].

All labels of HLPNGs are structural labels (see Sect. 3.4.2), which means that the user can edit them and leave them syntactically incorrect. Of course, this will not pass validation; but, it is possible to save nets with incorrect labels and load them again, so that the labels can be corrected another time.

PNML does not define or mandate a concrete syntax for declarations and terms. The concrete syntax for the labels is up to the tool. Therefore, the ePNK comes with its own concrete syntax, which resembles the one of CPN Tools [9], but is not identical! Before going into the details of the syntax, we briefly discuss some examples.

[11] Version 0.9.1 of the ePNK, did not support this feature yet.

[12] A ground-term is a term that does not contain variables.

[13] Actually, ISO/IEC 15909-2 would allow that this label is missing. This does not make much sense though, since in most cases there is no reasonable standard interpretation if the label is missing.

The following shows several declarations of variables, sorts and operators. Each of them could be in a separate declaration label, but they could also be contained in a single declaration:

```
vars
  x:NAT;

sorts
  A = MS(BOOL);

ops
  f(x:INT, y:INT) =  x * y,
  g() = 1;

sorts B = (A*INT), C = (B*B);
```

First, a variable x of built-in sort NAT is defined. Then a user-defined sort A is defined, which is a multiset over the built-in sort BOOL. Then, two named operations are defined, f and g. The operation f takes two parameters of type INT; the operation g does not have parameters. Note that named operations, basically, are abbreviations and, therefore, do not allow any recursion (see [6] for details). In the end, two other user-defined sorts are defined: B is a product of A and the built-in sort INT, and C is a pair over sort B. Note that also for sort declarations, recursion is not allowed.

The right-hand sides of the sort declarations above give you an idea of the syntax for sorts already. There are some built-in sort like BOOL, INT, NAT, POS and DOT. From these, we can built products or multiset sorts.

Here are some examples of terms (using the above declarations):

```
x‘f(x,x) ++ 1‘x ++  x‘g() ++  1‘5

1‘(dot,1) ++ 1‘(dot,1*1)

x > 1 and x < 5
```

The first is a multiset term over the sort INT, which could be used in arc inscriptions (if the attached place is of type INT). The second is a ground term over the product of built-in sort DOT with INT, where DOT is a sort that represents a type with a single element dot. The last term is a term of sort BOOL, which could be a condition.

The precise syntax is defined by the following grammar (that actually is a simplified version of the grammar that was used for generating the

parser). The terminals ID, INT, NAT, STRING in this grammar represent legal identifiers and legal representations of integer numbers, non-negative integer numbers and string constants.

Listing 3.1 shows the part of the grammar for declarations. Listing 3.2 shows the part of the grammar for terms. Note that we have simplified the grammar for making it more readable. The simplification, however, makes the grammar ambiguous (i. e. some texts could be parsed in two or more different ways). The ambiguities can be resolved again by assigning a binding priority to the different operators – moreover all operators are left-associative. Each line in the declaration of BinOp represents operators on the same level of priority, where the first line has the least binding-power and the last the highest. The unary operators (actually there is only one) have the highest binding power of all. Note that there are also some operators like the cardinality, which use circumfix notation: if m is some multiset `|m|` denotes the cardinality of that multiset. This operator has the same binding power as parentheses.

Listings 3.3 and 3.4 show the part of the grammar for built-in sorts and constants. Note that every number constant will implicitly be assigned the tightest fitting sort: INT, NAT, or POS. If a positive integer, say 5 should have the type INT instead, this can be expressed by 5:INT, which works like a type cast in object-oriented programming languages.

In addition to these syntactical constraints, the terms must also be correctly typed, which we do not discuss here in detail.

For HLPNGs, there are many constraints. Like for PTNets, arcs may only run from places to transitions or from transitions to places. All of the other additional constraints concern the correctness of the labels of HLPNGs. The following list gives an overview:

1. Every place must have a (correct) type.
2. Every declaration must be syntactically correct and correctly typed.
3. Every declaration must properly resolve (must not be recursive and all symbols it refers to must be defined).
4. Every term (in markings, arc annotations, and conditions) must be syntactically correct and correctly typed.
5. The marking of a place must be a ground term and must be a multiset over the sort of the place.
6. The arc annotation must be a term that is a multiset over the place's sort.

Listing 3.1: Grammar for declarations

```
Declarations :
    ( 'sorts' SortDecl ( ',' SortDecl )* ';'                    |
      'vars'  VariableDecl ( ',' VariableDecl )* ';'            |
      'ops'   OperatorDecl ( ',' OperatorDecl )* ';'            |
      'sortsymbols' ArbitrarySort ( ',' declaration )* ';'      |
      'opsymbols' ArbitraryOperator ( ',' ArbitraryOperator )*
    )*;

SortDecl :
    NamedSort | Partition;

NamedSort :
    ID '=' Sort;

VariableDecl :
    ID ':' Sort;

OperatorDecl :
    NamedOperator;

NamedOperator :
    ID '(' ( VariableDecl ( ',' VariableDecl )* )? ')' '=' Term;

Sort :
    BuiltInSort | MultiSetSort | ProductSort | UserSort;

MultiSetSort :
    'MS' '(' Sort ')';

ProductSort :
    '(' ( Sort ( '*' Sort )*)? ')';

UserSort :
    ID;

ArbitrarySort :
     ID;

ArbitraryOperator :
     ID ":" ( Sort ("," Sort )* )? "->" Sort;
```

Listing 3.2: Grammar for terms

```
Term :
    Term BinOp Term |
    UnOp Term       |
    BasicTerm;

BinOp :
    // all binary operators are left-associative
    'or' | 'implies'                      |  // lowest priority
    'and'                                 |
    '>' | '>=' | '<' | '<=' | 'contains' |  // all comparison ops
          '<r' | '<=r' | '>r' | '>=r' |     // on same level
          '<p' | '>p'  |                    //
          '<s' | '<=s' | '>s' | '>=s'     |  //
    '==' | '!='                           |
    '++' | '--'                           |
    '`'                                   |
    '+' | '-'                             |
    '*' | '**' | '/' | '%'                 ;  // highest priority

UnOp :
    'not' ;     // higher priority than all binary operators

BasicTerm :
    Variable                 |
    UserOperator             |
    OtherBuiltInOperator     |
    BuiltInConst             |
    '(' Term ')'             |  // a sub-term in parentheses
    '(' Term ( ',' Term )+ ')';  // a tuple

Variable :
    ID;

UserOperator :
    ID '(' ( Term (',' Term )* )?  ')' ;

OtherBuiltInOperator  :
    '|' BasicTerm '|' |  '#(' Term ',' Term  ')'  |
    CyclicEnumsBuiltInOperator  | PartitionsBuiltInOperator   |
    StringsBuiltInOperator | ListsBuiltInOperator;
```

Listing 3.3: Grammar for sorts and constants (1)

```
BuiltInSort  :
    Dot | Boolean | Number | FiniteEnumeration | CyclicEnumeration |
    FiniteIntRange | StringSort | ListSort ;

BuiltInConst  :
    DotConstant | BooleanConstant | MultisetConstant |
    NumberConstant | FiniteIntRangeConstant |
    StringConstant | ListConstant ;

MultisetConstant  :
    'all' ':' Sort  |
    'empty' ':' Sort;

Dot :
    'DOT';

DotConstant :
    'dot';

Boolean :
    'BOOL';

BooleanConstant :
    'true' | 'false';

Number :
    'INT' | 'NAT' | 'POS' ;

NumberConstant :
    INT (':' Number)?;
```

Listing 3.4: Grammar for sorts and constants (2)

```
FiniteEnumeration : 'enum' '{' ID ( ',' ID)* '}' ;

CyclicEnumeration : 'cyclic' '{' ID (',' ID)* '}' ;

CyclicEnumsBuiltInOperator :
     'succ' '(' Term ')'   |   'pred' '(' Term ')' ;

FiniteIntRange : '[' INT '..' INT ']' ;

FiniteIntRangeConstant : INT FiniteIntRange ;

Partition :
     'partition' Sort 'in' ID
       '{' PartitionElement ( ';' PartitionElement )* '}';

PartitionElement : ID ':' Term  ( ',' Term )* ;

PartitionsBuiltInOperator : 'partition' ':' ID '(' Term ')';

StringSort : "STRING" ;

StringsBuiltInOperator :
     "concatstring" "(" Term "," Term ")"      |
     // note that we do not have append (does not make sense)
     "stringlength" "("  Term  ")"             |
     "substring" ":" NAT "," NAT "(" Term  ")" ;

StringConstant : STRING ;

ListSort :  "LIST" ":" Sort;

ListsBuiltInOperator :
     "concatlists" "(" Term "," Term ")"               |
     "appendtolist" "(" Term "," Term ")"              |
     "listlength" "("  Term  ")"                       |
     "sublist" ":" NAT "," NAT "(" Term  ")"           |
     "memberat" ":" NAT "(" Term  ")"                  |
     "makelist" ":" Sort "(" (Term ( "," Term)* )? ")"  ;

ListConstant : "emptylist" ":" Sort ;
```

7. Every condition must be a term of sort BOOL.
8. Every declaration should have a distinct name (actually, this causes a warning only since this is a condition on concrete syntax, which is not part of PNML).
9. The parameters of every operation declaration should have distinct names (actually, this causes a warning only since this is a condition on concrete syntax, which is not part of PNML).

Note that PNML and ISO/IEC 15909-2 do not define a concrete syntax for declarations and terms. The syntax defined here is a syntax specific to the ePNK. In principle, a PNML document with a high-level Petri net in it could leave all the textual parts of the labels empty. In that case, the most important structure and content of these labels would not be visible in the graphical editor at all. The user would not see and would not be able to edit the labels textually. In order to convert this structural information into some text that can be edited by the user in the ePNK, a simple extension to the ePNK is deployed as a separate feature called "HLPNG Label Serialisation". It is recommended to install this feature.

If you have the "HLPNG Label Serialisation" feature installed, you can serialize all structured labels to the textual syntax of the ePNK. To this end, right-click on the respective HLPN element (the Petri net) in the tree-editor; then select "ePNK"→"Serialise HLPNG Labels".

3.6 Functions and Applications

The ePNK in its basic version does not come with much functionality for analysing, simulating and verifying Petri nets. Its main purpose, is to provide a graphical editor for Petri nets and PNML Documents, and to provide an infrastructure so that new Petri net types and new functions and applications for Petri nets can be plugged in. By and by, some functions have been developed that are now deployed together with the ePNK. And there are some applications of the ePNK, which are projects in their own right – for example the ECNO project, which generates code for so-called ECNO nets [16]. We hope, that over time, more functions and applications of other ePNK developers could be deployed together with the ePNK.

In this section, we explain some of the basic functions and applications that are deployed together with the ePNK. In these examples, we will also explain the *ePNK applications view*, which is used as a general user interface for the end user to control ePNK applications. Later, in the developers'

guide in Sect. 4 we will explain how you can contribute your own functions and applications to the ePNK.

Generally, the ePNK distinguishes *functions* and *applications*. A *function* is something, which is initiated on some Petri net, possibly asking the user for some extra input, then does some computation, and when the computation is finished, provides some output to the user in form of some dialog. After that, the function and its result are gone. One example of such a function is the verification of some CTL formula for some Petri net by a model checker, which will be discussed in Sect. 3.6.1. In particular, the model checker does not show or visualize any result in the Petri net itself. Also an *application* is initiated on some net. In contrast to a function, the application has a longer live-time, it shows some feedback to the user on top of the Petri net in the graphical editor, and also provides means for the user to interact with the application. An example of such an application is a simulator for high-level Petri nets, which will be discussed in Sect. 3.6.3. Here, the user will be shown graphically, which transitions are currently enabled, and by clicking on them, the user can determine which transition should fire next.

3.6.1 A simple model checker for EN-systems

In this section, we briefly discuss how to use a model checker, which can be initiated on P/T-nets. Note that even though this model checker is initiated on P/T-nets, the model checker interprets the net as an Elementary Net System (EN-systems) [28, 25], which means that at any time on any place, there can be at most one token. A transition that would add another token to a place, would not be able to fire – and arc-inscriptions are ignored.

Figure 3.9 shows a P/T-net which consists of several pages. Page pg0 (not visible) contains a single place semaphor with one initial token and pages pg1, pg2, and pg3 model three agents with the same life-cycle, which is shown in the graphical editor in Fig. 3.9. The three agents are competing for the semaphor in order to access their critical section criticalx. Actually, this net was automatically generated by a wizard for creating a net with any number of agents. This wizard can be initiated by "File"→"New"→"Other..." and then selecting "Multi-agent Mutex Net Wizard" from the category "ePNK".

The model checker on this net can be initiated by right-clicking on the Petri net element in the tree editor and then selecting "ePNK"→"Model checker". Then a dialog like the one in Fig. 3.10 pops up, where two CTL formula, which make sense in any system, are provided as a default input to the model checker:

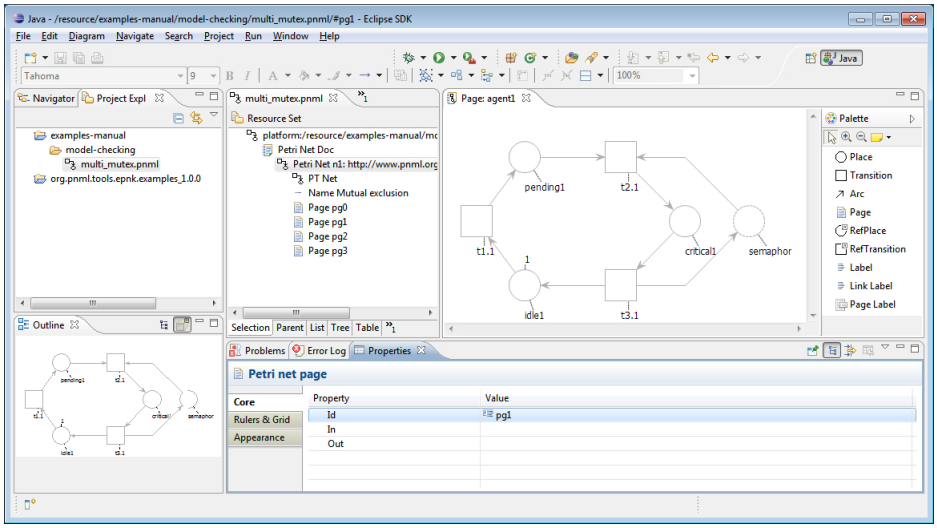

Figure 3.9: Mutex example with multiple agents

```
AG EX true, EG EX true # deadlock free, infinite path
```

As indicated by the comments behind the hash symbols, these two formula will check whether the system is deadlock free and whether there is at least one infinite path. You can of course enter some other formulas, which can

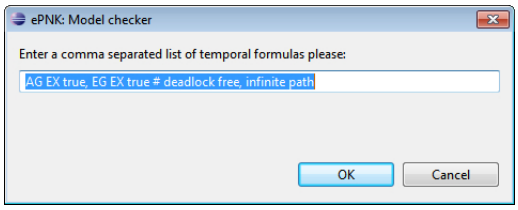

Figure 3.10: Model checking dialog: Input of formulas

use place names in order to formulate some more specific properties like:

```
AG !(critical1 and critical2) # mutual exclusion for 2 agents
AG (pending1 -> AF critical1) # ag.1: pending leadsto critical
```

For the exact syntax of the temporal formulas (CTL formulas), we refer to the documentation of the MCiE library `http://www2.cs.uni-paderborn.de/cs/kindler/Lehre/MCiE/` and its example formulas, and we recommend to have a look into the documentation of MCiE's parser package. Place

names will be used as variables in the formula. You need to make sure that places in the Petri net have legal MCiE variable names (in particular, there should not be white spaces or special characters in them). One speciality of the syntax of CTL formulas of MCiE is that the binary temporal operators, such as EU and AR, are represented in infix notation like $p1EUp2$ instead of the more common notation $E[p1Up2]$. Moreover, you can use the hash symbol `#` as a line comment – everything following the hash symbol in the same line is ignored by the MCiE parser.

If the formulas entered to the dialog in Fig. 3.10 are syntactically incorrect, the dialog will pop up again, indicating the position of the syntax error. You can either correct the error or abort the dialog.

If the sequence of formulas is syntactically correct and the dialog was not aborted, the model checker will be started on the net and check all the formulas. Since model checking can take quite some time for bigger nets, the actual model checking is done in the background, so that the Eclipse GUI is not blocked while the model checking is done. This is actually an Eclipse concept, which is called *jobs*. If a job should take too long, it can be aborted in the Eclipse *progress view*, which can be easily opened while jobs are running in the background by clicking on the *progress indicator* in the bottom line to the right of the Eclipse workbench[14].

When the model checking job is finished, this will be indicated also by a symbol in the bottom right corner of the Eclipse workbench. When you click on it, a dialog with the model checking result will pop up. For the net and the CTL formulas from the input dialog above, the result dialog is shown in Fig. 3.11.

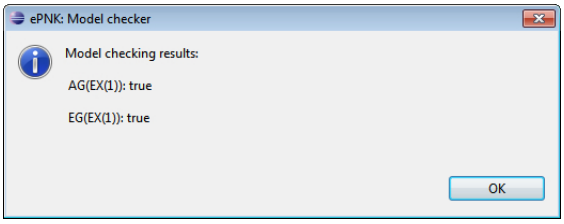

Figure 3.11: Model checking dialog: Result

[14]If the Eclipse progress area and icon are too small for you, you can open the Eclipse progress view explicitly: "Window"→ "Show View"→ "Other..." and then select "Progress" in category "General".

3.6.2 Applications view

As mentioned above, ePNK applications can be a bit more involved, since the user can interact with them, and applications can show some visual feedback to the end user and interact with the end user with visual feedback on top of the Petri net shown in the graphical ePNK editor. One example of such an application is an interactive simulator for high-level nets, which will be discussed in Sect. 3.6.3.

In order to explain the *application view* of the ePNK that is used for controlling the running applications and for choosing which application the user wants to interact with, we discuss a simple example of an application first. This example, computes the context of each transition, i. e. the places in the preset and the postset of a transition and the arcs between them. Obviously, this is not a too exciting application, but it serves its purpose to explain the application view of the ePNK, and serves as a simple example on how to implement applications later in the developers' section.

Figure 3.12 shows the ePNK with the nets from the model checking section again. Now, the transition context application is started, and the ePNK applications view is open – and the transition context application is selected. This application can be started on all types of Petri nets by right-clicking on the Petri net element (in the tree editor), and then selecting "ePNK"→"Start Transition Context App".

Each application has a current state. Depending on the type of state the application is using, this state might be associated with a visual feedback on top of the graphical Petri net. In the case of the transition context application, the states of the application are a list of all transition contexts; each state is a transition context consisting of a set of Petri net elements. The user can go back and forth in this list of transition contexts. Figure 3.12 highlights the context of transition `t2.1` in red. The red text in the top-left corner of the graphical editor shows to which application this graphical feedback belongs. The name "test" was freely chosen by entering it to the first column of the application view; entering a specific name here might be helpful once you have many different applications running on the same net. Note that the application view shows the feedback of only one application at a time. Which one is shown can be chosen by the user by clicking on the respective row of the application view. In our example from Fig. 3.12, there is only one application running, which is also selected. Applications can have their own specific actions, which will be shown in the top right corner of the applications view – once this application is selected. By default, an application has a *forward* and a *backward* action. In the case of the

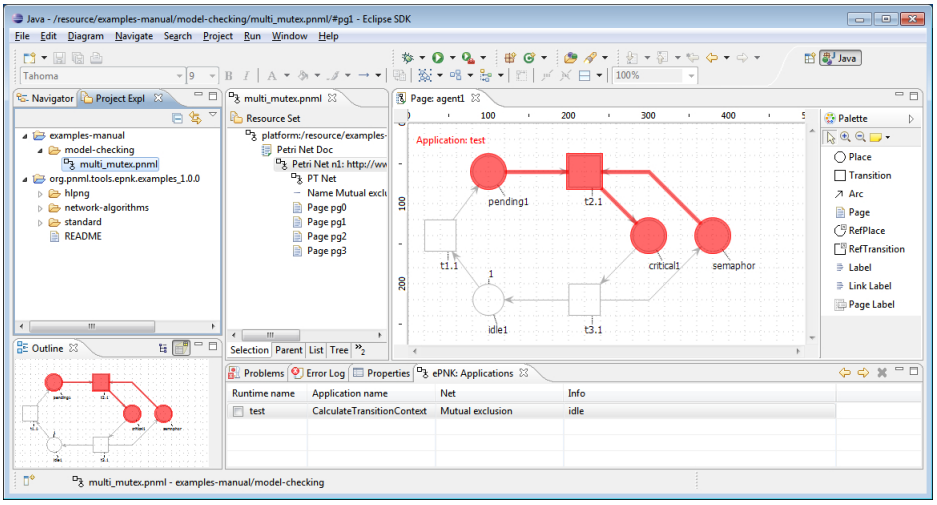

Figure 3.12: Application view with running transition context application

transition context, these buttons allow you to go back and forth in the list of transition contexts which will then be high-lighted in the graphical editor.

Note that you might not see all graphical feedback, since some pages are not open in the graphical editor or the graphical editor is not on the top. You need to open the pages on which you want to see the feedback yourself.

You can quit a running application, by checking the check box at the left of the row for this application, and then clicking on the delete button in the applications view. You can also select more than one application and, this way, quit more than one application at a time.

Initially, the application view is not opened. You can open it in the following way: Choose "Window"→"Show View"→"Other..."; then, in the opened "Show View" dialog select "ePNK: Applications" from the "ePNK" category.

Note that each application is attached to a net and the editor it was started from. When you close an editor, all applications running on nets of this editor will be automatically closed too.

3.6.3 A simulator for high-level nets

In this section, we discuss the simulator for high-level Petri nets, which is deployed together with the ePNK. It was developed by Mindaugas Laganeckas as part of his master's project [21]. The simulator is able to simulate high-level Petri nets as well as so-called *high-level net schemas* [18, 19, 23, 6, 8],

which can be instantiated with some communication network in order to simulate a network algorithm on a specific network [24].

Note that all examples that are discussed in this section can be obtained from the ePNK home page together with release 1.0.0 of the ePNK: `http://www2.imm.dtu.dk/~ekki/projects/ePNK/release-1.0.0.html`.

3.6.3.1 The basic simulator for high-level nets

We start with explaining the simulator for normal high-level nets in this subsection and explain the simulator for net schemas later in Sect. 3.6.3.3.

Figure 3.13 shows the simulator application running on a simple high-level net. The high-level net models a simple algorithm that computes the prime numbers according to the principle of the "Sieve of Eratosthenes": It starts with a multiset of all the numbers from 2 up to some upper limit (11 in our example) on place numbers. Then, the transition t removes a number (the value of x*y) from this place, if this number is a multiple of some other number (the value assigned to x) on that place. When no number on the place is a multiple of another number on that place, the transition cannot fire anymore – and the algorithm terminates. The numbers that are left on place numbers are prime numbers. In Fig. 3.13 there is only one last non-prime number left: 6. The current marking of the place is shown as a blue label at the top-right corner of the place (a long stack of tokens represented as a multiset term in the concrete syntax for HLPNGs of the ePNK).

Assuming that you have obtained and installed the examples from the ePNK home page, we will now explain how to start the simulator and how to open the additional simulator view, which shows the firing sequence up to the latest point in the simulation. Then we will explain how to interact with the simulator.

If you did not do that yet, you need to open the *ePNK applications view* as described[15] in Sect. 3.6.2. The *Simulation view* can be opened in a similar way: Choose "Window"→"Show View"→"Other..."; then, in the "Show View" dialog, select "Simulation View" from the "HLPNG Simulator Category". By clicking on the tab at the top of the views, you can arrange them in a way similar to Fig. 3.13 – it is convenient to see the applications view and the simulation view at the same time.

The simulator can be started on a high-level net by right-clicking on the HLPNG element in the ePNK tree editor and then selecting "ePNK"→"Start Simulator App".

[15] In short: Choose "Window"→"Show View"→"Other..."; then, select "ePNK: Applications" from the "ePNK" category.

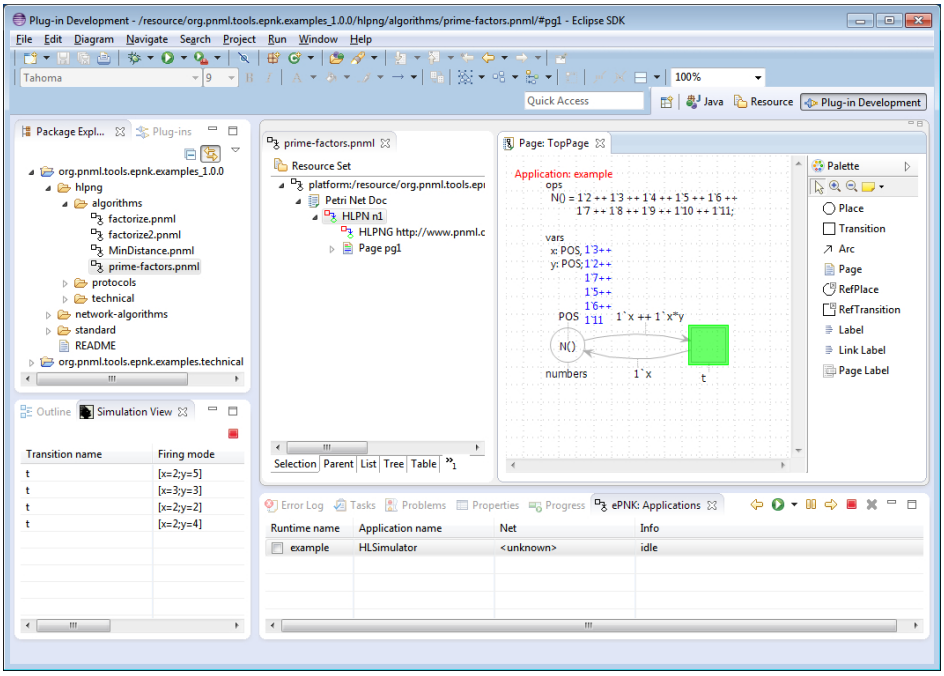

Figure 3.13: ePNK with a simulation application running on a HLPNG

Once the simulator application is started and selected in the applications view, the simulation view shows the firing sequence of all transitions (along with the firing mode) from the initial marking up to the last step of the current simulation. If no simulation application is selected, the simulation view shows the firing sequence of the last active simulation. You can click on the different entries and navigate up and down with the resp. buttons of the keyboard; then the net will show the marking before the selected transition fired. The current marking for each place is shown as a blue label at the top right of every place – if there is no label, the place's marking is currently empty.

When a simulator application is selected in the applications view, you will find several action buttons on the top right of the applications view, which can be used to control the simulator (as shown in Fig. 3.13). The *back* (left arrow) and *forward* (right arrow) buttons allow you to navigate back and forth in the firing sequence (which had been simulated already). The *play button* (white triangle in green circle), starts the automatic and random firing of some transitions – as long as there are enabled transitions. The *simulation speed* can be selected by a drop down menu on the small triangle

right of the play button. It can actually be changed while the automatic simulation is running. The automatic simulation can be stopped – actually "paused" – by the pressing the *pause button*. The automatic simulation can be started any time by pressing the play button again. By pressing the *stop button*, (red box), the simulator is reset to the initial marking as defined by the net. The cross to the right will delete the simulator completely (actually this is the general control for all applications).

The automatic simulation will randomly choose any of the currently activated transitions, and randomly choose a firing mode. If the simulation is paused, however, the activated transitions are high-lighted by a green overlay. You can click on these green transitions[16]; then, a menu will pop up, which shows all the possible firing modes for that transition from which you can select. Then, the transition will be fired in the selected mode. Clicking somewhere else or pressing the ESC button, will cancel the selection though.

Note that if you go back to some earlier state of the simulation by selecting a transition in the simulator view or by the back button in the simulator application, the marking at that point in the firing sequence will be shown. You will see that one transition is high-lighted by a blue (and darker) overlay. This is the transition to fire next in the firing sequence as shown in the simulation view. There might also be some other transitions high-lighted by a green overlay, which would have been alternative choices at that point. Note that also a blue transition might "hide" alternative choices that are not graphically high-lighted, since it might be able to fire it in different firing modes.

If you are in such an intermediate state of a firing sequence, you can still interact with the transitions by clicking on them as discussed above and by selecting a firing mode, which will fire this alternative transition in that marking. Note that in this case, all the later firing steps of the earlier firing sequence are deleted. From the current point on, a new branch of simulation will be followed. This way, you will be able to explore different branches of the reachability graph of the Petri net.

Note that, in some cases, the simulator is not able to compute the firing modes fully automatically, and is not able to decide whether a transition is enabled. In that case, the respective transition is high-lighted with a grey overlay. This does not happen in the prime factors example, but it will happen all-over in the example "factorize". Once you click on one of the transitions with a grey overlay, you will be prompted for possible values for

[16] You will also be able to click on a transition which is high-lighted in grey or blue, as will be discussed later.

the different variables. You can enter a semicolon separated list of values for each of the variables; then the simulator will try to compute possible firing modes based on these values. Note that you do not need to provide values for all variables; in many cases, it is enough to provide the value for one variable from which the values for the other variables can be derived. If the simulator can compute some enabled firing modes, the transition will be high-lighted in green, so that you can actually select the mode in which this transition should fire. You can still select "Manual input" for providing more or other possible values. If no modes could be found, the transition will remain high-lighted in grey; only if you provide values for which the transition can fire (or enough information for the simulator to figure that out), the transition will actually become enabled.

3.6.3.2 Supported operations

Note that the simulator is still in an experimental phase. In particular, some of the more specific operations of ISO/IEC 15909-2 are not supported yet. This, in particular, applies to the sorts and operators for symmetric nets which currently are not supported by the simulator.

The following sorts of ISO/IEC 15909-2 are supported in the current version (0.1.2) of the simulator: DOT, BOOL, NAT, POS, INT, and STRING as well as the generic sorts product, multiset and lists over existing sorts.

The following operators are supported: == and != on all sorts; or and and on BOOL; +, -, *, /, %, <, and > on NAT, POS and INT; concatstring for Strings; ', ++, --, all, and empty on multisets; the tuple operator for products; emptylist, makelist, memberat, sublist, length, appendtolist, and concatlists for lists.

The sorts and operators introduced for symmetric nets are not supported by the simulator at all.

3.6.3.3 The simulator for network algorithms

In this section, we discuss the simulator for network algorithms. Before discussing the simulator itself, we discuss the concept of network algorithms and the way they are modelled as algebraic nets schemas or – in the terminology of ISO/IEC 15909 – high-level Petri net schema (HLPNGS). To this end, we use an example which – except for syntactic sugar – is almost identical to the first publications that used algebraic net schemas for modelling and verifying network algorithms [18, 30, 23]: a simple algorithm that, for a given network of computing agents with some distinguished root agents,

computes the minimal distance of each agent from a root agent. The Petri net modelling the algorithm is shown in Fig. 3.14. In the example projects

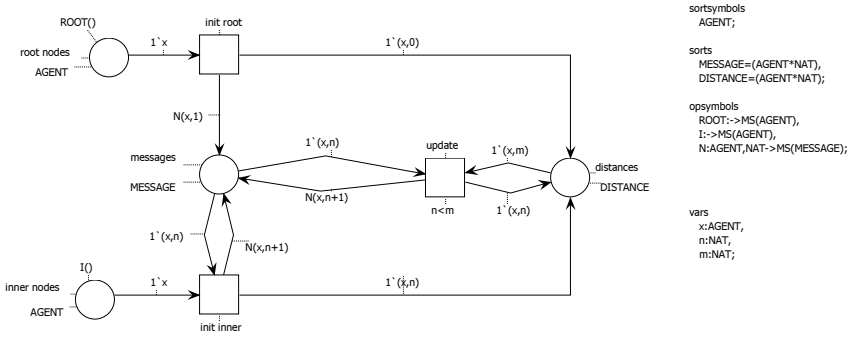

Figure 3.14: Network algorithm computing the minimal distance to a root

deployed for the ePNK 1.0.0, you will find it in subfolder min-distance of folder network-algorithms.

One of the main features of a Petri net schema is that, the Petri net model itself is completely independent from the actual network of agents on which the algorithm is working. In the model from Fig. 3.14, the set of agents of the network is represented by the sort AGENT, but it is just a symbol – which still needs some interpretation. The multiset constant ROOT represents the set of root nodes, and the constant I represents the set of non-root nodes (or inner nodes).

Moreover, there is an operation N, which takes an AGENT and a natural number as a parameter. This function represents sending a message from one agent to all its neighbours in the network – where the message itself is a natural number. A MESSAGE to an agent is represented by a pair, where the first component is the receiver AGENT and the second component is the actual content of a message. If there is a distance computed for some agent already, this is also represented as a pair of an AGENT and a number NAT – for making the difference clear, we call this pair DISTANCE.

Initially, the place root nodes contains all the root nodes of the network; the place inner nodes contains all the inner nodes. The transition init root models the initial step of a root node x: it sets its own distance to 0, which is represented as a pair $(x, 0)$, and adds a message to all its neighbours that they might have distance 1 from a root node – all these messages are represented by $N(x, 1)$. Transition init inner models the initial step of an inner node: when an inner node x initially receives some message with some distance n, it stores this distance as a potential shortest distance, and sends

a message to all its neighbours with distance $n+1$, which is represented by $N(x, n+1)$. An inner node x can later receive other messages with another distance n; if the other distance n is less than its current distance m, the agent takes distance n as its new distance, and sends a message with distance $n+1$ to all its neighbours. This is modelled by transition update.

As said before, the Petri net model from Fig. 3.14 models an the minimal distance algorithm for any network. If we want to simulate the algorithm, we need to know on which network it should run. To this end, a very simple network editor is deployed together with the high-level simulator of the ePNK. Figure 3.15 shows the network editor with a simple example network. In this case, it is a network with directed arcs. The network simulator can

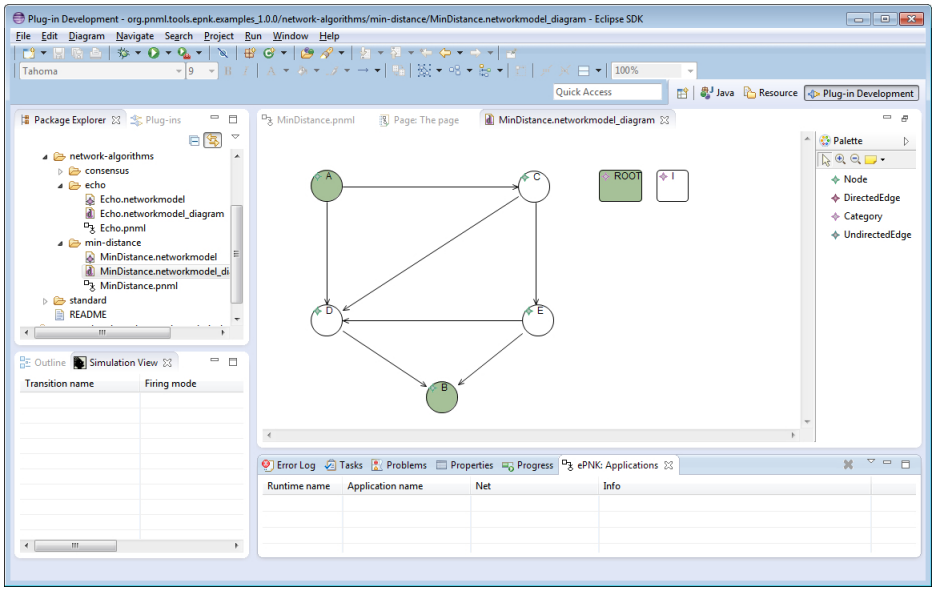

Figure 3.15: A network on which the algorithm could work

be started by right-clicking on the HLPN element in the ePNK tree editor and then selecting "ePNK"→"Start Network Simulator App"; if there is a network file with the same name as the Petri net model in the same folder, the network simulator chooses this network for the simulation. If there is no such file, the user will be prompted for a file with a network model. Once the network model is selected, the interpretations of the sort AGENT, the constant symbols ROOT and I, as well as the function N (sending a message to all the neighbours of the agent) are defined by this network. For example, for the network of Fig. 3.15, the set associated with the sort

AGENT is $\{A, B, C, D, E\}$; the constant ROOT denotes the multiset $[A, B]$, the constant I denotes the multiset $[C, D, E]$; for $x = A$ and $n = 5$, the term N(x,n) will evaluate to the multiset $[(C, 5), (D, 5)]$ – representing the message 5 to each neighbor of agent A. And these will be the interpretations the simulator will be using for simulating the net.

Figure 3.16 shows the network simulator running on the minimal distance algorithm from Fig. 3.14 on the network from Fig. 3.15. The interaction with

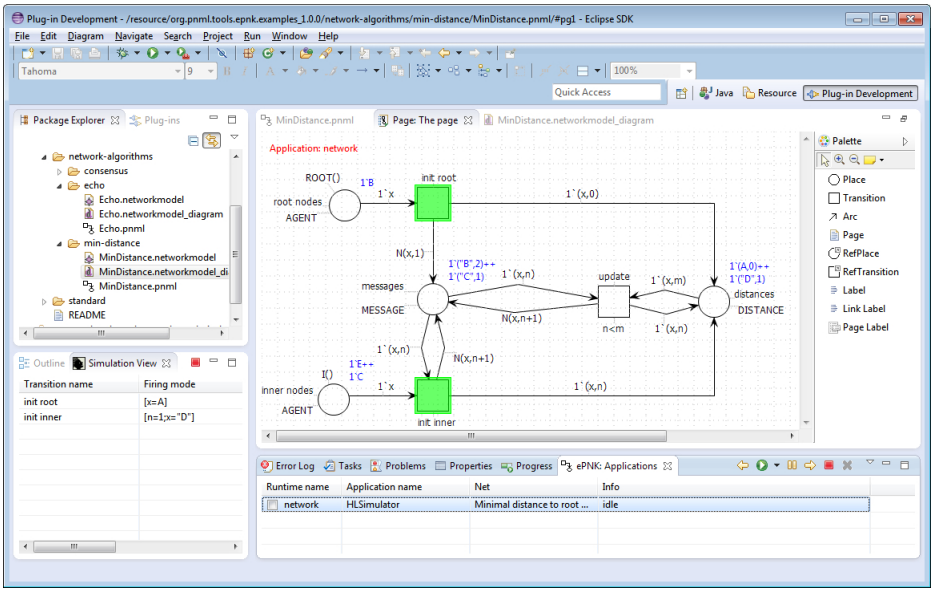

Figure 3.16: Network simulator running on the minimal distance algorithm

the simulator and the way to control the simulation is exactly the same as described for the basic simulator in Sect. 3.6.3.1, once the network simulator is properly initialized with a network.

The editor for the *network* is a simple *editor* generated by GMF and follows the GMF philosophy. A new network can be created by "File"→ "New" → "Other..." and then selecting "Network Diagram" from category "Examples". In this diagram, you can add nodes, directed and undirected edges between the nodes, as well as categories for nodes. When a node is selected, each node can be associated with any number of categories by a menu that pops up when clicking in the category property in the properties view. In Fig. 3.15, the association with categories is also shown by the same colour for the category and the resp. nodes; but the colour itself does not have any meaning in the diagrams.

For a given network, the sort AGENT is associated with the set of nodes; for every category, the respective constant symbol is associated with the multiset of nodes that are in that category (in our example these are the root nodes and the inner nodes). For the operation symbol N, $N(x, m)$ denotes a multiset of pairs, where for each outgoing arc from x to y there is a pair (y, m) in the multiset. $S(x)$ is a multiset of pairs over agents, where there is a pair (y, x) in the multiset if there is an undirected arc from x to y; these represent all messages that are sent from agent x to all its neighbours. Likewise $R(x)$ contains all the messages received from all its neighbours.

A Petri net modelling the echo algorithm, which is another example deployed together with the ePNK, makes use of the send and receive operations S and R. But, we do not explain the echo algorithm here – see [30, 19] for details[17].

3.7 Limitations and pitfalls

The current version of the ePNK (1.0.0) still has some minor limitations, which are discussed in this section in order to avoid some unpleasant surprises. If you identify other issues or problems, please, let us know.

3.7.1 Saving files: Tree editor

Technically, the tree-editor and the graphical editor for pages are different editors. The graphical editors can be initiated via the tree editor only (via the pop-up menu "Start GMF Editor on Page" or a double click on a page in the tree editor or the graphical editor). The tree editor, however, serves as the main editor; saving a PNML document is possible only from the tree editor – and only the tree editor shows a valid "dirty-flag" when the file contains unsaved changes (see Sect. 3.3.3)

3.7.2 Reset an attribute

In case some attribute is set to some value and you would like to remove that value (meaning that actually, the attribute is removed completely), it is no good to enter an empty string as a value for that property. And if it is an attribute with finitely many possible values, the drop down menu does not give you an option to not select any value. To restore the default situation of unsetting the attribute, you can right-click into the property column of

[17] The operation S is denoted with M and the operation R is denoted with $\overline{M}$ in [19] – and the other way round in [30].

that attribute in the properties view and then select "Restore default value" (see Sect. 3.4.3).

3.7.3 Graphical features

The graphical editor of the ePNK supports many graphical features, such as positions and size of nodes, intermediate positions for arcs, fonts, size and colours for labels, colours and line-width for nodes and arcs. Up to now, not all of these graphical features are transferred back to the actual PNML document (some of this information it is not even supported by PNML). Some graphical information is stored only as (ePNK) tool specific information.

The graphical information that is used and made available in PNML so that it can be used by other tools that support PNML is the following:

- Position and size of nodes.
- Line colour and background colour for nodes.
- Image information for nodes[18].
- Position resp. relative position of labels[19].
- Text colour[20] and background colour for labels.
- The font and the font-size of labels.
- Intermediate points of arcs.
- Line colour of arcs.
- The shape attribute of arcs (straight or bezier).

Note that all the graphical information – even the one that is not transferred to PNML features – is saved by the ePNK as tool specific information. You can even put notes on the graphical canvas, which will be saved. The graphical information not covered in the above list, however, will not be

[18]Up to version 1.0.0 of the ePNK, the path to an image can be changed only in the tree editor. But the image is shown in the graphical editor once it is set. From version 1.0.1 and higher, it will be possible to set the image URI in the properties view of the node directly

[19]The size of labels is not supported by PNML

[20]For the text colour the ePNK is actually abusing the line colour attribute of PNML, since PNML does not have a text colour attribute.

available to other tools, and will be gone, when the tool specific information (org.pnml.tools.epnk.diagraminfo) for the resp. page is deleted.

The ePNK reads all graphical attributes that are supported by PNML; but, only the ones discussed above are used and shown in the graphical editor. Moreover, the unused graphically attributes are not checked for validity; they will be written exactly the same way they were, when loading the PNML file (see also Sect. 3.7.6).

Due to some limitations in the automatically generated GMF editor on which the graphical editor of the ePNK for pages is based, there is another quirk of the graphical ePNK editor: When a node is moved, the intermediate points of the attached arcs are also changing in the diagram; these changes are, however, not propagated to the PNML model. If you want to make sure that the intermediate points of an arc of the diagram and the PNML are exactly the same, the you need to explicitly touch and slightly move an intermediate point of each attached arc – only then, the exact positions of the intermediate points are properly propagated to the PNML model.

Note that some Petri net type definitions might define some graphical appearance for some of its elements. In that case, this graphical appearance overrides the graphical attributes of PNML.

3.7.4 Petri net types

As mentioned in Sect. 3.3.2, the type of a Petri net should never be changed after a net of that type was created – unless you know exactly what you are doing. Otherwise, it could happen that the produced PNML is invalid.

For HLPNGs, it is no problem to change its kind any time, since this kind has an effect on the validation only, but no effect on the serialisation of the net to a PNML file. Changing the "kind", actually, does not change the Petri net type – just the sub sets of features that are supported.

3.7.5 Wrapping labels

All labels in ePNK can have line-breaks. In the graphical editor, line-breaks can be added by pressing the CNTRL and ENTER at the same time.

3.7.6 Graceful PNML interpretation

PNML files that are produced by the ePNK and which have been successfully validated are conformant to PNML as defined in ISO/IEC 15909-2. The only exception is, when some illegal graphical attributes are read from an existing PNML file; these attributes will not be touched by the ePNK, and therefore

written again – even if they are not conformant to ISO/IEC 15909-2. But, if a PNML file is created by using the ePNK only and if it validates correctly, the saved file is PNML conformant.

The ePNK, however, is not a PNML validator. It reads PNML and "PNML-like" documents and writes them again in a graceful manner. This way, it is possible to save PNML documents that do not properly validate; and the ePNK is able to load these files again even though other PNML tools might not be able to load them. For example, when some elements do not have ids (as required by PNML), references to these elements cannot be made via their id. In that case, the ePNK uses XPath references to these elements, which is not conformant to PNML. If such an invalid PNML file is loaded later by the ePNK again, the ids are added then, and validation is successful, saving this file will produce a conformant PNML documents again.

3.7.7 Deviation from PNML

There is one minor deviation of the ePNK from PNML as of ISO/IEC 15909-2:2011: In the ePNK, all declarations of HLPNGs can have a name attribute, which comes from the fact that `Declaration` implements the interface for symbol definitions (`SymbolDef`). As a consequence, also the `Unparsed` declaration can have a name attribute. In ISO/IEC 15909-2, `Unparsed` does not have a name attribute – as the only exception among all declarations of PNML. Therefore, if an `Unparsed` operation declaration with a name attribute would be manually added to a PNML document in the ePNK tree editor, the resulting PNML document would not be conformant to ISO/IEC 15909-2:2011 anymore.

In practice, however, this should not be any problems, since the only way to add an unparsed declaration to a HLPNG net would be to add it manually via the ePNK tree editor. If all declarations are edited in the graphical editor by editing the respective labels, this problem will never arise. If the PNML document would contain an `Unparsed` operation declaration without a name attribute (as it would be according to the standard), the ePNK would show this as an error under validation. But, it would still be able to read and write the PNML document.

In future versions of ISO/IEC 15909-2[21], this might be resolved by requiring that all declarations – including `Unparsed` – should or, at least, could have a name attribute.

[21] Currently ISO/IEC 15909-2:2011 is under revision, which might fix this problem in the standard.

Chapter 4

Developers' guide

In this chapter, we discuss how to extend the ePNK, with new functionality, with new Petri net types, with new graphical appearances, or with new tool specific extensions. For all these extensions, the ePNK provides *extension points* so that the extensions can be made without changing the actual code of the ePNK[1]. Actually, the ePNK does not even provide own extension points for adding functionality: The existing Eclipse extension points are good enough for that for now[2].

Section 4.3 shows how to add some functionality to the ePNK, which could be a model checker, or some other analysis or verification function for Petri nets, or which could be a function that reads a net in PNML and produces some net in some other format, or a function that generates a net that is stored in PNML format. Section 4.4 shows how to implement applications that high-light the results in the graphical editor and can interact with the end user while they are running.

Section 4.5 shows how to add new Petri net types to the ePNK. Simple net types can almost completely be generated from a model; for more complex Petri net types, such as high-level Petri nets, a mapping to XML must be defined and parsers need to be implemented.

Section 4.6 shows how to define a customized graphical appearance for some graphical features of Petri nets.

Section 4.7 shows how to add tool specific information to the ePNK.

[1] Technically, you would not even need to see the code of the ePNK, but looking at it might help understand the ideas and principles behind the ePNK.

[2] Eventually, there might be some ePNK-specific extension point for adding functionality for getting a more uniform "user-impression" of the added functionality. Since this is mostly a GUI-issue, this does not have the highest priority right now.

At last Sect. 4.8 gives an overview of the projects of the ePNK and Sect. 4.9 briefly discusses how to deploy own extensions.

All of these extensions are discussed by the help of some examples, which are deployed together with the ePNK. In these examples, we assume that the reader is familiar to the main principles and ideas of Eclipse, its plug-in architecture, and Eclipse plug-in development. We cannot give a detailed introduction to Eclipse and to Eclipse plug-in development here (when you have the feeling that you need more background on these issues, the "Platform Plug-in Developer Guide" which is part of Eclipse might be a starting point: "Help" → "Help contents"; and there are many other Eclipse resources [27, 4]). Still, Sect. 4.1 gives a brief overview of Eclipse plug-in development. We go a bit more into the details for EMF and explain some of the steps that need to be done in EMF more explicitly, but it is also recommended to read up on some details on EMF [2] before starting with own development projects.

Section 4.2 gives a brief introduction to the PNML core model in the ePNK, its differences to the PNML core model from ISO/IEC 15909-2; some basic understanding of the API generated from this model is crucial for using the ePNK as a developer.

4.1 Eclipse: a development platform for the ePNK

As briefly discussed in Sect. 3.1 already, Eclipse is an Integrated Development Environment (IDE). Here, we briefly explain how to set up the Eclipse environment so that you can work on your own extensions and do the tutorials of this chapter – assuming that you have installed Eclipse and the ePNK as explained in Chapter 1 already.

4.1.1 Importing ePNK projects to the workspace

As a developer, you would probably want to have a look at some parts of the source code of the ePNK – though not strictly necessary. Therefore, the ePNK is deployed in such a way that you can easily import the relevant ePNK plug-in projects to your workspace[3] with their source code and their

[3]Unfortunately, there are some projects where the source code was – accidentally – not included to the deployed version, or the configuration of the class path was not properly set up for that purpose. These plug-ins will be made available on the ePNK installation page for version 1.0.0 – as far as they relevant for this manual. And we will try to fix that in future versions.

models, so that you can inspect the code and the models from which major parts of the code were generated.

Here, we explain how to import the ePNK projects into your development workspace as source projects. Initially, it is recommended to import only the basic project `org.pnml.tools.epnk` into the workspace; later during this developers' guide, we will mention several other projects that might be interesting for you to have a look at. In the end, Sect. 4.8 gives an overview of important ePNK projects and their main function and purpose.

In order to import an ePNK project (or any other plug-in project which is running behind the scenes), you first need to open the Eclipse *Plug-ins view*. To this end, select in the Eclipse workbench "Window" → "Show view" → "Other..."; then select "Plug-ins" from the category "Plug-in Development". Typically, this view would also open when you switch to the *Plug-in Development perspective* of Eclipse. Once the Plug-ins view is open, browse this view and find the ePNK plug-in `org.pnml.tools.epnk`. Right-click on this plug-in and select "Import As" → "Source Project". After that you will find the project `org.pnml.tools.epnk` in the Eclipse package explorer or some other open resource browser.

In this project, you can see the source code and also the model files in folder "model". But, if you did not install some extra tools yet, you will not be able to open these models with some reasonable editor. Section 4.1.2 discusses which additional tools you need to install to your version of Eclipse so that you are able to inspect – and later create – these kind of models and diagram files.

You can import all ePNK projects to the workbench as discussed above, but only some few projects will be relevant for you. This chapter introduces you to the most relevant ones – one after an other (in the end, Sect. 4.8 gives an overview of the ePNK projects). If you, eventually, are confident in developing functions and applications for the ePNK, you might also want to contribute to the ePNK and make your extensions and changes to the ePNK plug-ins. In that case, you can ask to be given access to the ePNK development repository.

In your workbench, you have now the project `org.pnml.tools.epnk`, which is the basis for developing new plug-ins and, in particular, extensions to the ePNK. We start calling this workbench *development workbench* now. The reason for introducing an additional adjective to the term *workbench* here is that you need to start another workbench from this one, in order to start Eclipse with the new extensions that you are developing in the development workbench. This additional instance of Eclipse is called the *runtime workbench* since this is when the ePNK with your new extensions

is running. It will very much look like the original ePNK as discussed in Chapter 3 – just with the extensions from your development workbench also running. The runtime workbench can be started from the development workbench by "Run" → "Run Configurations..." and then selecting "Eclipse Application" and pressing the "New" icon and then "Run". After you have started the development workbench in this way once, it will be enough to press the "Run" button in the tool bar for starting the runtime workbench again. For now, however, we do not start the runtime workbench since we need to implement some new functionality first.

4.1.2 Installing the EMF and Ecore Tools SDK

As mentioned already, a major part of defining a new Petri net type is creating an Ecore model which captures the concepts of the new Petri net type; most of the code can then be generated from such a model. In order to be able to do that, you need to install two additional extensions to your Eclipse: the "EMF Modeling Framework SDK" and the "Ecore Tools SDK".

To install these extensions to Eclipse, start the "Install dialog"[4] select the standard Eclipse update site and select "EMF Modeling Framework SDK" and "Ecore Tools SDK" and follow through the installation process.

After restarting Eclipse, you can check whether the installation was successful: Open the project `org.pnml.tools.epnk` and, in that project, open the folder `model`. The files with extensions ".ecore", ".ecorediag", and ".genmodel" should have special icons now. When you open "PNMLCoreModel.ecorediag" by double-clicking on it, you should see an Ecore model in a class diagram like graphical notation as shown in Fig. 4.1.

4.2 The PNML core model in the ePNK

For using the ePNK as a developer its is important to have a more detailed understanding of the *PNML core model* since the API for accessing, navigating, and modifying Petri nets is generated from this model. In Sect. 4.1.2, you have seen already were to find and how to open the diagrams of the PNML core model and some related diagrams. And when you start developing with the ePNK, you will probably need to have a look into these diagrams once in a while, in order to understand the details about the relation between the different classes and concepts of PNML.

[4]Remember: In order to start the install dialog, select "Help" → "Install New Software..." and then choose `http://download.eclipse.org/releases/juno` as the standard update site (in this case Juno).

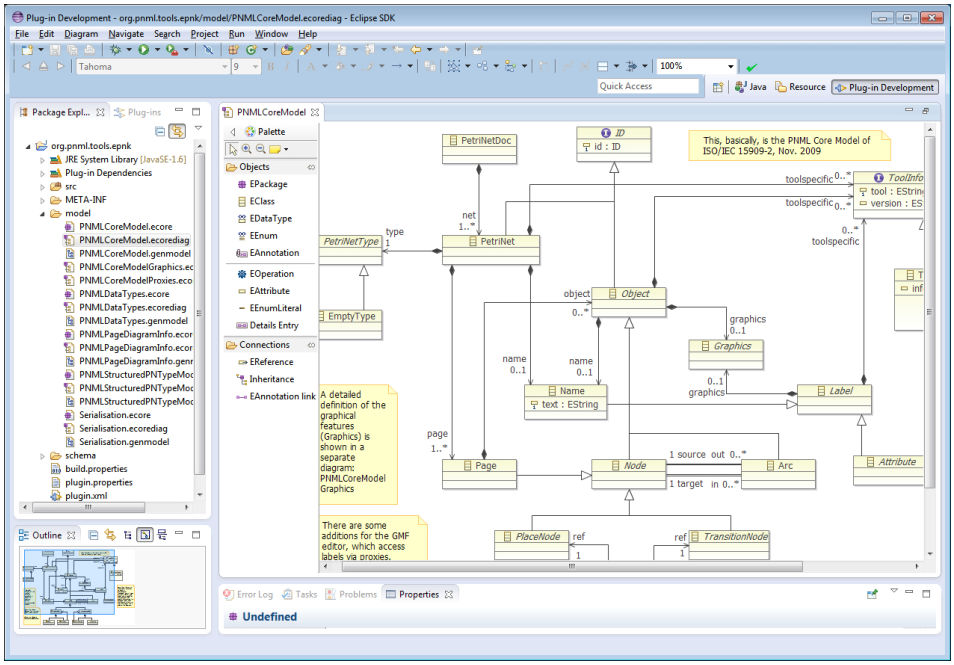

Figure 4.1: Developer's view with the ePNK PNML core model

In this section, we give an overview of the PNML core model of the ePNK and discuss some of the differences to the PNML core model of ISO/IEC 15909-2. As discussed in Sect. 2.2.1 already, the PNML core model of the ePNK is slightly more general than the one of ISO/IEC 15909-2:2011 [8] (see Fig 2.1). One of the main differences is the following: According to ISO/IEC 15909-2, a page is not considered to be a node; in the ePNK, a page is considered to be a node. This way, it is possible to define Petri net types in the ePNK that allow arcs to be connected to pages (e. g. in order to define a Petri net type that mimics substitution transitions of CPNs [9]). Keeping this in mind, might be particularly important, when defining the constraints for arcs of net types (such as the ones for P/T-nets as defined in Sect. 4.5.1.4), when you do not want to allow to connect pages to other nodes with arcs.

The other differences of the PNML core model are a bit more technical and will be discussed below. You might want to have a look at Fig. 2.1 of the PNML core model of ISO/IEC 15909-2:2011 and at Fig. 4.1, while we discuss the difference – or open the diagrams in the ePNK for seeing even more details.

Most importantly, the `type` of a Petri net is an attribute of the class `PetriNet` according to ISO/IEC 15909, whereas, in the ePNK, it is a composition to a class that must inherit from the abstract class `PetriNetType`. Instances of this class represent the Petri net type and will provide some services to the ePNK to generically maintain the features of the respective Petri net type. This is discussed in more detail in Sect. 4.5. The PNML core model of the ePNK also provides one concrete class for a Petri net type, which does not exists in ISO/IEC 15909-2:2011: `EmptyType`, which represents a Petri net without any labels other than `Names`.

In the ePNK, the `source` and `target` reference of the class `Arc` have a corresponding opposite reference from the `Node` to its `out`-going and `in`-coming `Arc`s. These opposite references are not serialized to the XML document, however, since this would not be compliant with the PNML format. The opposite references will, however, be restored when loading the Petri net. These opposite references are very convenient when navigating between different elements of the net.

In turn, the ePNK does not have a class `Annotation` of ISO/IEC 15909-2, which represents those *labels* that should be shown as graphical annotations of the respective node or arc in the graphical editor. There exists a class `Attribute`, however, which represents those labels that should be represented as properties of the respective element only, but not as annotation in the graphical editor[5]. In the ePNK, all *labels* that are not attributes are considered to be annotations. This has historical reasons, since the first version of the ePNK did not support attributes at all; but, it also avoids making mistakes in Petri net type definitions: it is impossible to define labels that are neither attributes nor annotations.

At last, the PNML core model of the ePNK defines a separate interface `ID`, which is used to unify all those elements that need a unique identifier in a PNML document. All classes that must have such an identifier, implement the interface `ID` in the PNML core model of the ePNK. The reason for adding the class `ID` to the ePNK PNML core model is that, this way, the ePNK can handle elements with an identifier in a uniform way; with the separate id attribute for all elements as defined in the PNML core model of ISO/IEC 15909-2, this would require much more effort.

The attentive reader might also have noticed that the PNML core model of the ePNK does not contain any OCL constraints, whereas the PNML core model of ISO/IEC 15909-2 does. This, however, does not mean that

[5]By plugging in some graphical extensions, attributes can still have an effect on the graphical appearance of a Petri net.

the ePNK ignores these constraints; in the ePNK, they are just realized in a different way: as constraints that are plugged in on top of the model. We will see examples of how ePNK constraints are plugged in to the ePNK resp. to Eclipse later (e. g. in Sect. 4.5.1.4).

4.3 Adding functions

Next, we discuss how to add new functionality to the ePNK. As discussed earlier, there are two different kinds of new functionality. *Functions* take some net, possibly some user input, do some computing and then report a result – typically via some dialog window. After that, the function is over and done with. The model checker from Sect. 3.6.1 is a typical example of a function. By contrast, *applications* are started on a net; then, they show some feedback on top of the graphical editor, the user can interact with the application, and the visual feedback will be updated accordingly. Typically, an application finishes only when the user explicitly closes or quits it. The simulator for high-level nets of Sect. 3.6.3 is a typical example of an application.

In this section we, discuss the implementation of functions for the ePNK. Basically, we use the standard mechanism of Eclipse to plug in and start functions – as *views*, *wizards*, *actions*, *command handlers*, *jobs* or *dialogs*. In this manual, we explain the use of these concepts on the side, as far as they are necessary. Since setting up jobs in a proper way can sometimes be a bit tedious, the ePNK provides some utility classes that make programming jobs a bit easier.

The focus of this section is on the use of the API of the ePNK that is generated from the PNML core model and the Petri net type definitions by the *Eclipse Modeling Framework* (*EMF*). On the side, we point out some of the general principles of EMF and some practical issues on working with EMF. For a more detailed introduction to EMF, we refer to [2]. In this section, we do not only show how to access, navigate and manipulate nets; we discuss also how to open, create and write PNML files programmatically.

To this end, this section discusses how to plug in functionality into the ePNK (or to Eclipse in general) in three different ways.

- Section 4.3.1 shows how to implement a new Eclipse view, which gives an overview of a PNML file that is selected in one of the Eclipse resource explorer views. This includes opening and loading the contents from a PNML file.

- Section 4.3.2 shows how to implement a *wizard* for creating a PNML file (actually, we changed a wizard that was automatically created by the Eclipse "new plug-in project wizard"). This wizard creates a PNML file with a P/T-System that represents a simple mutual exclusion algorithm for a number of agents – where the number can be selected by the user in one of the dialogs of the wizard. The Petri net is split up to different pages, so that the Petri net for each agent is contained in a separate page. In particular, this example shows how to create a PNML file, fill it with some contents, and to save it.
- Section 4.3.3 shows how to implement a simple pop-up menu on a selected Petri net (in the tree editor), which starts a model checker, asking the user for some formulas to be checked, and then checking the formulas on the net. Since model checking can take quite some time, the model checker will run in the background and can be aborted by the user. This uses Eclipse's concept of jobs. On the side, this shows how to use some of Eclipse's user dialog functions.

In the end, Sect. 4.3.4 gives an overview of the different functions of the ePNK (and the API generated from its EMF model), some hints on how to work with this API in Eclipse and in EMF general, as well as some additional ePNK utilities and helper classes that make it more convenient to handle and access some of the information stored in a PNML document. Experienced Eclipse and EMF developers, might want to start reading the overview in Sect. 4.3.4 first, and come back to Sect. 4.3.1–4.3.3 for some details later.

4.3.1 Accessing a PNML file and its contents: A file overview

In this section, we discuss how to implement a new (and very simple) Eclipse *view*, that will give an overview of the contents of a PNML file that is selected in the explorer. Figure 4.2 shows a screenshot of the result. For the selected file "hlpng-gmf.pnml" in the "Project Explorer", the "ePNK File Overview" in the bottom left, shows that the selected file is a Petri net document, which contains 3 Petri nets, a high-level net, a P/T-net, and an empty net; the type of a net is represented by its unique URI. The name of the first net is "A high-level next example"; the other two nets do not have a name.

This view and its functionality is implemented in the plug-in project `org.pnml.tools.epnk.functions.tutorial`. We go through this project now[6]. In addition to the overview view discussed above, this project also

[6]Remember that you can import the source code of that project to your development

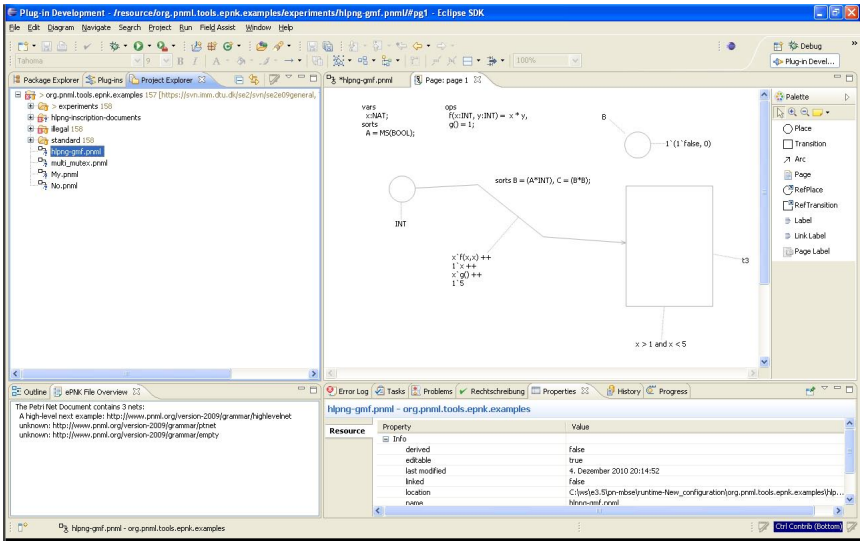

Figure 4.2: The ePNK with the "File Overview" view

contains the implementation of the wizard for creating a PNML document which will be discussed later in Sect. 4.3.2.

The implementation of the view is contained in a single Java class: `PNMLFileOverviewView` in the package `org.pnml.tools.epnk.functions.tutorial.overviewview`. We briefly explain the general structure of this view class, an extract of which is shown in Listing 4.1; we deleted all imports, an attribute definition, and all comments, which can be looked up in the source code. The class extends the Eclipse `ViewPart`, which actually makes it an Eclipse view, and it implements the `ISelectionListener`, which allows our view to obtain the information on the current selection of the user in the workbench. Note that this class does not have an explicit constructor. The reason is that the view will be set up via the `createPartControl()` method: In the first three lines of that method, a viewer (which represents the content of that view) is initialized, and so-called providers will enable the view to properly show the contents. Since these are standard Eclipse providers, we do not discuss the details here. In the last two lines of the `createPartControl()` method, our viewer registers itself with the Eclipse selection mechanism as a *selection listener* and then creates the information that should be shown for the current user selection. We will discuss the

workspace from the "Plug-ins" view by selecting the project, right-clicking on it and then choosing "Import As" → "Source Project".

Listing 4.1: Class `PNMLFileOverviewView`: Infrastructure

```
package org.pnml.tools.epnk.functions.tutorials.overviewview;

import ...

public class PNMLFileOverviewView extends ViewPart
  implements ISelectionListener {

  ...

  private TableViewer viewer;

  public void createPartControl(Composite parent) {
    viewer = new TableViewer(parent);
    viewer.setContentProvider(new ArrayContentProvider());
    viewer.setLabelProvider(new LabelProvider());
    getSite().getPage().addSelectionListener(this);
    selectionChanged(null, getSite().getPage().getSelection());
  }

  public void setFocus() {
    viewer.getControl().setFocus();
  }

  public void dispose() {
    super.dispose();
    getSite().getPage().removeSelectionListener(this);
  }

  public void selectionChanged(IWorkbenchPart part,
      ISelection selection) {
    if (selection instanceof IStructuredSelection) {
      IStructuredSelection structured =
        (IStructuredSelection) selection;
      Object first = structured.getFirstElement();
      if (first instanceof IFile) {
        viewer.setInput(getOverviewInfo((IFile) first));
  }   }   }

  public String[] getOverviewInfo(IFile file) { ... }

}
```

respective method `selectionChanged()` below. Note that there are two other methods. The `setFocus()` methods forwards the focus properly to the content of the view, once the view is focused. More important is the `dispose()` method: the implementation of this method makes sure that our view removes itself as a selection listener once it is disposed (which typically would happen when the user decides to close the view).

Once the view has registered itself as a selection listener with the Eclipse workbench, its `selectionChanged()` method is called whenever there is a change in the user's selection. In the implementation, of this method, the kind of the current selection is analysed and it is checked whether the first selected element is a file (i. e. whether it implements the interface `IFile`). If so, the method `getOverviewInfo()` for accessing the actual contents of the file and for computing the contents of the overview is called; this produces an array of Strings, which then will be set as the new contents of that view – and, this way, shown to the user.

The `getOverviewInfo()` method is probably the most interesting part here, since it shows how to open and access a PNML or a PNX file (we do not even need to make a difference). The implementation of this method is shown in Listing 4.2. Up to line 7, it is checked whether the file extension is either "pnml" or "pnx" (the two file extensions, the ePNK uses for storing Petri net files – "pnml" represents files in PNML format, and "pnx" represents Petri net files in XMI, which we also call PNX); then, the path to that file is extracted and a URI of that path is created. Actually, Eclipse provides also user dialogs and file dialogs that would allow us to ask the user for a file name that would be returned as a URI; here we used the selection mechanism and the file to get hold of some legal URI of a PNML or PNX file. Therefore, the code that comes now, could be used at any other point when a program wants to read and access some file, once we have a String with the path to the file: This starts with creating a *resource set* and, within that resource set creating a *resource* with the given URI, which is the first parameter of the `getResource()` method; the second parameter indicates whether cross-references to other resources should be resolved lazily or not (which is not relevant here). Note that in EMF, a resource or file should always be accessed (and created, see Sect. 4.3.2 for more information) in this way: via a resource set. After we successfully got the resource, we can obtain its contents by the `getContents()` method which returns a list of its top-level objects – in case of PNML, this list should contain exactly one element. Being defensive, we check whether the contents exists and whether its first element is an instance of PetriNetDoc. If so, we go systematically through all the contained nets, get their names and their PNML types and

Listing 4.2: Class `PNMLFileOverviewView`: Accessing the file

```
public String[] getOverviewInfo(IFile file) {
  String[] result = {"No ePNK file selected"};
  String extension = file.getFileExtension();
  if (extension != null &&
      (extension.equals("pnml" ) || extension.equals("pnx"))) {
    String path = file.getLocationURI().toString();
    URI uri = URI.createURI(path);

    ResourceSet resourceSet = new ResourceSetImpl();
    Resource resource = null;
    try {
      resource = resourceSet.getResource(uri, true);
    } catch (Exception e) {
      result[0] = "File could not be read.";
      return result;
    }

    List<EObject> contents = resource.getContents();
    if (contents != null && contents.size() > 0) {
      EObject object = contents.get(0);
      if (object instanceof PetriNetDoc) {
        PetriNetDoc document = (PetriNetDoc) object;
        List<PetriNet> nets = document.getNet();
        int no = nets.size();
        result = new String[no + 1];
        result[0] = "The Petri Net Document contains "
          + no + (no == 1 ? " net" : " nets:");
        no = 1;
        for (PetriNet net : nets) {
          String name = net.getName() != null ?
              net.getName().getText() : "unknown";
          String type = net.getType() != null ?
              net.getType().toString() : "unknown";
          result[no++] = "  " + name + ": " + type;
        }
      } else result[0] = "The file does not contain a PetriNetDoc.";
    } else result[0] = "The file does not contain any element.";
  }
  return result;
}
```

add a String with that information to the String array with the result. In the other cases, we return some error messages. Note that we do not need to close the file, or do anything else after we have obtained the information we need.

Let us have a closer look at how the contents of the Petri net document is accessed, once we have obtained a `PetriNetDoc` object. For any reference and attribute of the *PNML core model*, the ePNK API has corresponding *getter* and *setter methods*. For example, if a class has an attribute `name` of type `String`, the `getName()` method will return the `String` with that name, and with `setName()` we could set it – but we do not change the net in this example. For attributes and features with a multiplicity greater than one, this is slightly different. For example, a PetriNetDocument can contain many nets; therefore, `getNet()` will return a list of nets, which then is iterated over to get the individual nets. And by adding a net to this list, this Petri net would be added to the Petri net document (see 4.3.2 for examples).

As stated above, class `PNMLFileOverviewView` implements the "ePNK File Overview" as we have seen it in Fig. 4.2. But, it is not enough to just implement this class; it would not show up in Eclipse, because Eclipse would not know that it exists. In order to make the view known to Eclipse, we need to define it as an *extension*: This is done in the project's "plugin.xml" file. Double clicking on the "plugin.xml" file, will give you a convenient editor for defining and editing the extensions you want to define. Explaining the actual extensions is a bit easier with the XML fragment that is produced by this editor. The fragment relevant for our overview view is shown in Listing 4.3. It says that we contribute our *extension* to the Eclipse *extension point* `org.eclipse.ui.views`, which is a new Eclipse view. The category defines where and under which category the new view can be found, when the user wants to open it via the Eclipse "Show View" menu. We define a category specifically for the ePNK and then define the actual view referring to this category. The attributes of the view say that a view of this kind can at most be open once, that it uses the above category, and refer to the class which actually implements it: `PNMLFileOverviewView`; and the attributes define an icon (used in the tab of that view) and a name for that view.

Note that in order to access some of the classes like `Resource`, `Resource Set`, and some of the ePNK classes like `PetriNetDoc`, `PetriNet`, etc. in the implementation of the view, we would also need to define dependencies to the plug-in projects by which they are provided: If you open the file "plugin.xml", you will find these projects in the tab "Dependencies". But this is a more technical issue, which we do not go into the details.

Listing 4.3: Defining the overview view extension in "plugin.xml"

```
<extension
      point="org.eclipse.ui.views">
   <category
         name="ePNK"
         id="org.pnml.tools.epnk.views.category">
   </category>
   <view
         allowMultiple="false"
         category="org.pnml.tools.epnk.views.category"
         class="org.pnml. ... .overviewview.PNMLFileOverviewView"
         icon="icons/PetriNetDoc.gif"
         id="org.pnml.tools.epnk.extensions.tutorials.pnmloverview"
         name="ePNK File Overview">
   </view>
</extension>
```

Now, you could start the runtime workbench; from there, you could open the "ePNK File Overview" by "Window"→ "Show View"→ "Other..." and then selecting "ePNK File Overview" in the category "ePNK". This will show the view in the workspace as shown in Fig. 4.2. Whenever the user selects some PNML or PNX file in the package explorer, this view will show an overview of the contents of the selected file. Since this plug-in project is deployed with the ePNK 1.0.0 already, it is of course not even necessary to start the runtime workbench – the view is already there in the development workbench, once the ePNK is installed.

4.3.2 Writing PNML files: Generating multi-agent mutex

Next, we discuss how to create new PNML files and how to fill them with some contents. In typical applications, the contents might come from a file in a format of another Petri net tool, which should be converted to PNML. In our example, however, we programmatically generate a Petri net: my favourite semaphore mutex example, which was used in Sect. 3.6.1 as an example for model checking already. To make things slightly more interesting, the number of agents competing for the semaphore is a parameter. This function is implemented as an Eclipse *wizard* and it was implemented by creating a new file wizard for PNML files automatically by the Eclipse "New Plug Project" wizard choosing the "custom plug-in

wizard" with the choice of the "New File Wizard" in the "Template Selection" dialog. But, this does not need to bother you too much. If you are interested in the manual changes made to the automatically generated code, you will find all the manual changes in the two classes in the package `org.pnml.tools.epnk.functions.tutorials.wizards` enclosed by comments like `// eki: ...` . These packages are contained in the same plug-in project, `org.pnml.tools.epnk.functions.tutorials`, as discussed in Sect. 4.3.1 already.

In the rest of this section, we focus on the explanation of the parts of the implementation that are concerned with the creation of the file and its contets. This functionality is implemented in the method `createPNMLFile()` of the class `MultiAgentMutexNetWizard`, which is shown in List. 4.4. The parameter `path` is a String representation of a path to the file that should be created. The parameter `number` is the number of agents that should be created in the mutex net that will be generated.

Creating the Petri net programmatically is quite straightforward, but code intensive. Therefore, we have split up the creation process into several parts for the different elements, which will be discussed top-down from creating the document, the net, its pages, and the places, transitions, reference places, and arcs on them. We discuss these methods one after the other – and omit some boring ones in the end (you can find all details in the source code). Listing 4.4 shows the method that creates the file: First, it calls the method `createPetriNetDoc()` that creates the Petri net document, which

Listing 4.4: Method `createPNMLFile(String path, int number)`

```
public void createPNMLFile(String path, int number) {
  PetriNetDoc doc = createPetriNetDoc(number);

  final URI uri = URI.createURI(path);
  ResourceSet resourceSet = new ResourceSetImpl();
  final Resource resource = resourceSet.createResource(uri);
  EList<EObject> contents = resource.getContents();
  contents.add(doc);
  try {
    resource.save(null);
  } catch (IOException e) {
    // Do nothing for now if file could not be saved.
  }
}
```

is discussed later. This is the contents of the file that we want to write. Then, we convert the path into a URI. Then, we create a resource set – from which the resource (the file) is create. Surprisingly enough, this is already all we need to do. At this point, we can add the contents to the resource. Note that it does not even matter whether the resource is a PNML file or a PNX file – Eclipse will, dependent on the file extension, chose the right implementation of the resource so that either a PNML file or a PNX file is written once we save the resource in the end. But, we configured the wizard in such a way that the user can chose only the "pnml" extension.

Adding the contents follows the same principle that we have discussed already. With `getContents()` we get a list of EMF objects (which would be empty, since the resource was newly created right now); then we add the Petri net document to this list. The only thing left is to save the resource, which is done by calling the `save()` method. Note that the save method has a parameter, that could be used to configure the way a file is saved. But, `null` is fine here – and you should only change this, if you know exactly what you are doing.

Let us dive a bit deeper into the method `createPetriNetDoc()`, which takes one parameter only – the number of agents. This method is shown in Listing 4.5. In the second line, a new Petri net document is created. Note that this is not done with the usual `new` construct of Java. Rather, the *factory* for the PNML core model which can be obtained by `Pnmlcoremodel Factory.eINSTANCE` is used for this purpose. It is part of the EMF philosophy that clients using the generated code should not know anything about the actual implementation of classes. And EMF strongly recommends to create new objects only via these factories. Note that the new net and its type are also created by factories – since the type is plugged in, the factory of that plug-in is used for this purpose: `PtnetFactory` resp. its instance. After creating the net, its id is set by the `setId()` method. Note that this could be any string, but it is our responsibility to make sure that all ids are different (if we chose to create the ids programmatically). Then, the net is added to the list of nets of that document: to this end, we get the list of all nets of the document via the `getNet()` method on which we call the `add()` method. There is no way to directly add a net to a document. The type of the net can, again, be set with the `setType()` method, since the type does not allow for multiple values.

After that, a name label is created, its text value is set, and the name label is added to the net.

Next, a new page is created by calling a separate method, which is added to the list of pages of the net, and the place semaphore is created as the

Listing 4.5: Method `createPetriNetDoc(int number)`

```
1  public PetriNetDoc createPetriNetDoc(int number) {
     PetriNetDoc doc = PnmlcoremodelFactory.
       eINSTANCE.createPetriNetDoc();

     PetriNet net = PnmlcoremodelFactory.eINSTANCE.createPetriNet();
6    net.setId("n1");
     doc.getNet().add(net);
     PetriNetType type = PtnetFactory.eINSTANCE.createPTNet();
     net.setType(type);

11   Name nameLabel = PnmlcoremodelFactory.eINSTANCE.createName();
     nameLabel.setText("Mutual exclusion");
     net.setName(nameLabel);

     Page page = createPage(type, "pg0", "semaphor page");
16   EList<Page> pages = net.getPage();
     pages.add(page);

     Place semaphor = this.createPlace(
       type, "semaphor", "semaphor", 1, 380, 140);
21   page.getObject().add(semaphor);

     for (int i=1; i<= number; i++) {
       page = createAgentPage(type, semaphor, i);
       pages.add(page);
26   }

     return doc;
   }
```

single object on this page. To this end, we use the `createPlace()` method.

In the for-loop at the end of the method, for each agent, there will be created one page with the net for each agent.

All the other methods follow the same principles, and there is not too much interesting to see in them. Therefore, we finish with discussing the method `createAgentPage()`, which is shown in Listing 4.6. This method creates 3 places, one reference place (referring to the semaphore that was created on the first page above), 3 transitions, and 8 arcs. What makes this method a bit more interesting is the graphical information that is added to the arcs: some intermediate point, which makes the net look a bit nicer. If you have a closer look at the `createTransition()`, `createPlace()`, and `createRefPlace()` you will find similar constructs for defining the position and size of the nodes, and the position of the labels associated with them. But this is straightforward and follow the exact principles of ISO/IEC 15909-2 (see [6]). Once the implementation of the wizard class is finished, it must be made know to Eclipse: it must be plugged in via the "plugin.xml". But, we do not discuss this here since this is similar to plugging in a view (have a look into the "plugin.xml", if you are interested).

In the runtime workbench (or a version of the ePNK in which this plug-in is installed already), you could invoke this function as follows: Go to the resource explorer – or any other explorer – of the workbench, press the right mouse button and select "New"→"Other...", select "Multi-agent Mutex Net Wizard" in the category "ePNK". Then, a dialog opens in which you can choose a folder[7] ("container") in which this file should be created, a "file name" (which must have extension "pnml"), and the number of agents. Note that, normally, the file creation wizard would overwrite existing files. This "multi-agent" wizard, however, was modified in such a way that existing files won't be overwritten accidentally.

4.3.3 Long-running functions: A model checker

In this section, we discuss the implementation of a model checker for P/T-Nets, which, actually, are interpreted as EN-Systems here. The model-checker is based on a simple library for symbolic model checking that was developed for teaching purposes: *Model Checking in Education* (*MCiE*)[8]. This MCiE library is deployed as part of the ePNK tutorials.

[7] If you have selected exactly one folder when you invoke the wizard, the fields of this dialog will be pre-set.

[8] see `http://www2.cs.uni-paderborn.de/cs/kindler/Lehre/MCiE/`

Listing 4.6: Method `createAgentPage()`

```
public Page createAgentPage(PetriNetType type, Place sem, int i) {
  Page page = createPage(type, "pg"+i, "agent"+i);

  Place idle = createPlace(type, "idl"+i, "idl"+i, 1, 100, 220);
  Place pending = createPlace(type, "pen"+i, "pen"+i, 0, 100, 60);
  Place critical = createPlace(type, "cri"+i, "cri"+i, 0, 300, 140);
  RefPlace semRef = createRefPlace("sem"+i, "sem", sem, 380, 140);
  Transition t1 = createTransition(type, "t1."+i, "t1."+i, 40, 140);
  Transition t2 = createTransition(type, "t2."+i, "t2."+i, 220, 60);
  Transition t3 = createTransition(type, "t3."+i, "t3."+i, 220,220);

  Arc a1 = createArc(type, "a1."+i, idle, t1);
  Arc a2 = createArc(type, "a2."+i, t1, pending);
  ...
  Arc a6 = createArc(type, "a6."+i, t3, idle);

  Arc a7 = createArc(type, "a7."+i, semRef, t2);
  Coordinate coordinate =
    PnmlcoremodelFactory.eINSTANCE.createCoordinate();
  coordinate.setX(300);
  coordinate.setY(60);
  ArcGraphics arcGraphics =
    PnmlcoremodelFactory.eINSTANCE.createArcGraphics();
  arcGraphics.getPosition().add(coordinate);
  a7.setGraphics(arcGraphics);

  Arc a8 = createArc(type, "a8."+i, t3, semRef);
  ...
  a8.setGraphics(arcGraphics);

  EList<Object> contents = page.getObject();
  contents.add(idle);
  contents.add(pending);
  ...
  contents.add(t3);
  contents.add(a1);
  ...
  contents.add(a8);

  return page;
}
```

In this developers' guide, we will not go into the details of model checking and its theoretical foundation, since this is not the point of this tutorial at all. For more information on model checking, we refer to a standard text book on model checking [3]. The point of this tutorial is to show how some function can be installed as a *pop-up* menu with an *action* on a Petri net that is open in the tree editor[9]. Model checking can actually be quite computation intensive and could take quite some time; therefore, we need to make sure that the actual model checking action does not block the graphical user interface of Eclipse while the model checker is running. Eclipse provides *jobs* for this purpose, which allow to run tasks (or jobs) in the background. The ePNK provides some convenience classes that make it a bit easier to set up and start jobs, which are running in the background – and provide a possibility to show a result in a dialog, once the job is finished.

The model checking functionality is implemented in the plug-in project `org.pnml.epnk.functions.modelchecking`. Like the other projects, you can import the source code of this project to your workspace via the Eclipse "Plug-ins" view and the "Import As" → "Source Project" menu. The actual model checker is implemented in the class `ModelcheckingJob` in package `org.pnml.epnk.functions.modelchecking.action`. The action initiating the model checking job is `ModelcheckingAction` in the same package.

Since the class `ModelcheckingAction` and the way it is integrated to the ePNK is quite simple, we start with explaining this one first. It is shown in Listing 4.7. This class extends the `AbstractEPNKAction`, which is an ePNK convenience class that makes it easy to add a new action, which initiates a job. The class `AbstractEPNKAction` overrides two methods: `isEnabled()` and `createJob()`. The method `isEnabled()` checks whether the action is applicable for the selected Petri net. In our example, it checks whether the Petri net has a type and whether this type is `PTNet`. The other method `createJob()` creates the actual job, which is an instance of class `ModelcheckingJob` with a Petri net and a defaultInput (a default formula for the user dialog in our case). This class extends the ePNK's convenience class `AbstractEPNKJob` and will be discussed later.

In order to make the action `ModelcheckingAction` know to Eclipse and appear in the popup menu in the "ePNK" category, we need to define an extension. Listing 4.8 shows the part of the "plugin.xml" file that defines this extension (with some ellipses).

[9]Note the extension points for pop-up menus and actions are deprecated in Eclipse 4.2. But, they still work – eventually, this will be adjusted by using Eclipse commands and handlers.

Listing 4.7: The action class `ModelcheckingAction`

```
package org.pnml.tools.epnk.functions.modelchecking.action;

import
    org.pnml.tools.epnk.actions.framework.jobs.AbstractEPNKAction;
import org.pnml.tools.epnk.actions.framework.jobs.AbstractEPNKJob;

import org.pnml.tools.epnk.pnmlcoremodel.PetriNet;
import org.pnml.tools.epnk.pnmlcoremodel.PetriNetType;
import org.pnml.tools.epnk.pntypes.ptnet.PTNet;

public class ModelcheckingAction extends AbstractEPNKAction {

  @Override
  public boolean isEnabled(PetriNet petrinet) {
    if (petrinet != null) {
      PetriNetType type = petrinet.getType();
      return type != null && type instanceof PTNet;
    }
    return false;
  }

  @Override
  protected AbstractEPNKJob createJob(PetriNet petrinet,
      String defaultInput) {
    return new ModelcheckingJob(petrinet,defaultInput);
  }

}
```

Listing 4.8: Defining the popup action for the model checking action

```
<extension
    point="org.eclipse.ui.popupMenus">
  <objectContribution
      id="org.pnml.tools.epnk.functions.modelchecking.contribution1"
      objectClass="org.pnml.tools.epnk.pnmlcoremodel.PetriNet">
    <menu
        id="org.pnml.tools.epnk.actions.standardmenu"
        label="ePNK"
        path="additions">
      <separator
          name="group1">
      </separator>
    </menu>
    <action
        class="org.pnml.tools.epnk. ... .action.ModelcheckingAction"
        enablesFor="1"
        id="org.pnml.tools.epnk.functions.modelchecking"
        label="Model checker"
        menubarPath="org.pnml.tools. ... .standardmenu/group1">
    </action>
  </objectContribution>
</extension>
```

Listing 4.9: The constructor of `ModelcheckingJob`

```
public ModelcheckingJob(PetriNet petrinet, String defaultInput) {
  super(petrinet, "ePNK: Model checking job");
  if (defaultInput != null) {
    defaultformula = defaultInput;
  }

  place2variable = new HashMap<Place,Variable>();
  place2primedvariable = new HashMap<Place,Variable>();
  transitions    = new Vector<Formula>();
  placeNames    = new HashSet<String>();
  duplicateNames = false;
}
```

At last, we have a look at the class `ModelcheckingJob`, which is implementing the user dialogs (asking the user for temporal formulas), converting the Petri net into ROBDDs, doing the actual model checking, and showing the result to the user again. In addition to the constructor, we need to implement (override) the following methods of `AbstractEPNKJob`): `prepare()`, `getInput()`, `run()`, `showResult()`, and `canceling()`. Below we explain the implementation of the constructor and the methods:

Constructor: Sets up all the data structures needed during the job; typically, this will be storing the default input. In our model checker example, we also set up some mappings, for mapping places of the Petri net to variables of the MCiE library, and mappings from transitions to formulas defining their behaviour, and some other information. The code of the constructor is shown in Listing 4.9.

`prepare()`**:** This method is handling the user dialogs before the actual job starts. In our case, it asks the user for some CTL-formulas; it also allows the user to correct the input, if the formulas are syntactically incorrect – or to abort the action. The code for this user dialog is shown in Listing 4.10. Since this is standard Eclipse programming, we do not go into the details of this part here. The only relevant part for the ePNK is that the job will not be continued, if the `prepare()` method returns false – in the implementation of the model checking job, this is done, when the user presses cancel in one of the dialogs (line 11/12 and line 33/34).

In our model checking job, the `prepare()` method will try to convert

Listing 4.10: The user dialog of the `prepare()` method

```
...

InputDialog dlg = new InputDialog(
    null,
    "ePNK: Model checker",
    "Enter a comma separated list of temporal formulas please:",
    defaultformula,
    null);
dlg.open();

if(dlg.getReturnCode()!=Window.OK)
  return false;

defaultformula = dlg.getValue();

do {
  try {
    Parser parser = new Parser(new StringReader(defaultformula));
    formulas = parser.parseFormulaList();
    parser.parseEnd();
  } catch (Exception e) {
    formulas = null;
    dlg = new InputDialog(
        null,
        "ePNK: Model checker",
        "Syntax error in formula: \n\r" +
        e.toString() + "\n\r" +
        "Fix the error please or press cancel:",
        defaultformula,
        null);
    dlg.open();

    if(dlg.getReturnCode()!=Window.OK) // Didn't click on OK!
      return false;
    defaultformula = dlg.getValue();
  }
} while (formulas == null);
```

Listing 4.11: Building the formula for the initial marking (in `prepare()`)

```
FlatAccess flat = new FlatAccess(getPetriNet());

init = new Constant(1);
for (org.pnml.tools.epnk.pnmlcoremodel.Place p : flat.getPlaces()) {
  if (p instanceof Place) {
    Place place = (Place) p;
    registerPlace(place);

    PTMarking marking = place.getInitialMarking();
    if (marking != null && marking.getText().getValue() > 0) {
      init = new BinaryOp(BinaryOp.AND,
          init,
          place2variable.get(place));
    } else {
      init = new BinaryOp(BinaryOp.AND,
          init,
          new UnaryOp(UnaryOp.NOT,place2variable.get(place)));
    }
  }
}
```

the Petri net into formulas defining the behaviour of the transitions and the initial marking. And on the way, it will be checked whether there are duplicate names of places, so that a warning can be issued. Listing 4.11 shows the part of the `prepare()` method converting the initial marking into a state formula. The basic idea is that, in this formula, a variable corresponding to the place occurs exactly once. It occurs negated, if the place is not marked and it occurs without negation, if the place is marked (with at least one token[10]). All these negated and un-negated variables are connected by boolean and-operations as formulas represented in MCiE's data structure. What is more interesting here is that the ePNK provides a way to access a net that consist of pages with reference nodes in a flattened way. This convenience class of the ePNK is called `FlatAccess`, which can be initialized with a Petri net of any type. Then, the instance `flat` is used to get all places of the net, independently of the pages they occur on. Likewise, `flat` provides methods to access all the transitions and to get all the input

[10]Remember that we abuse P/T-Nets for representing EN-Systems.

and output arcs of a place or transition (including the ones of the reference nodes referring to them). This way, it is easy to obtain the pre- and post-sets, without being bothered with the page structure. For some more examples of the use of these methods, you can have a look into the code that converts transitions into formulas, which however is not discussed here.

The last part of the prepare method, is converting the formulas into ROBDD-representation and creating a transition system out of these formulas. This is shown in Listing 4.12. Again, this is specific to MCiE. But, there are two parts that are important for the `prepare()` method in general: With `this.setName()`, we can give the job a specific name, which is used in Eclipse's jobs view. In our example, we say that it is a model checking job, add the name of the net and the formula which the user entered. The last important part is that the `prepare()` method returns `true` in order to indicate that the preparation successfully terminated, and the actual job can be run (in the background) now.

Listing 4.12: Finishing the `prepare()` method

```
Name name = getPetriNet().getName();
String netref = "";
if ( name != null && name.getText()!= null) {
  netref = " on net " + name.getText();
}

this.setName("Model checking job" + netref +": " + defaultformula);

Context context = new Context();
ROBDD is = init.toROBDD(context);
ROBDD ts[] = new ROBDD[transitions.size()];
ChangeSet css[] = new ChangeSet[transitions.size()];

for (int i = 0; i< ts.length; i++) {
  ts[i] = transitions.get(i).toROBDD(context);
  css[i] = new ChangeSet(context);
  transitions.get(i).addChangedVariables(css[i]);
}
transitionsystem = new Transitionsystem(context,is,ts,css);

return true;
```

Listing 4.13: The `run()` method

```
protected void run() {
  result = "Model checking results:\n\r";
  for (int i = 0; i < formulas.length; i++) {
    ROBDD obdd = formulas[i].toROBDD(transitionsystem);
    result = result + " " + formulas[i] + ": " +
      transitionsystem.isValid(obdd) + "\n\r";
  }
}
```

Note that all computations in the `prepare()` method should run fast. Computations that are time-consuming should be implemented in the `run()` method, which will be run in a separate thread in the background.

`getInput()` This method is called by the action, to get and store as default for the next call, the user input. In our case, the formula that was entered by the user during the prepare phase (in its String representation as entered by the user) is returned.

`run()` This method implements the part of the job that will be run in the background – and typically contains the computation intensive parts. In our case, this is the actual model checking task. The implementation of the method is actually quite simple (most of the programming work lies in the preparation). It is shown in Listing 4.13. Still, it is the most computation intensive part, which is why we are using the job to run it in the background. Note that, at the end of this method, we also prepare the `result` already in a String that will be shown to the user. But, there must not be any user dialog in the `run()` method itself, since this method is run in a separate thread in the background – and user dialogs would require to be called from a dedicated GUI thread.

`showResult()` This is the method that will be called for showing the result to the user. And, the infrastructure from `AbstractEPNKJob` will make sure that it will be called from the dedicated GUI thread again. Therefore, we can use all Eclipse dialogs for showing the result. Listing 4.14 shows the implementation of this method. The result String, which was prepared during the `run` method is shown to the user by initiating an information dialog.

Listing 4.14: Code for showing the final result

```
protected void showResult() {
  MessageDialog.openInformation(
      null,
      "ePNK: Model checker",
      result
  );
}
```

Listing 4.15: Code for aborting the model checking job

```
protected void canceling() {
  if (transitionsystem != null) {
    transitionsystem.abort();
  }
}
```

`canceling()` This method is a call-back mechanism that allows Eclipse – typically triggered by the end user – to abort a job that is running in the background. In the case of computation intensive jobs, to abort the computation and not to let that thread continue in the background is very important; otherwise this thread would consume all the computation power until it finishes on its own – which could take extremely long. Therefore, MCiE provides a mechanism for aborting model checking operations on some model, by invoking `abort()` – from a different thread of course. Our implementation of the `canceling()` method invokes this `abort()` method to actually terminate the model checking – possibly with some delay. This is shown in Listing 4.15, where `transitionsystem` is the one that was constructed before in the `prepare` method and on which the model checking is done. This will actually cause the model checker – the thread in which the computation is running – to throw some exception at some point of its computation in the `isValid()` method; this will stop the complete thread, since the exception is not caught.

Together, the classes `ModelcheckingAction` and `ModelcheckingJob`, plugged in via the "plugin.xml" implement a simple, but complete model checker. If the model checking project was not already part of the ePNK, you would start the runtime workbench, and would have the model checker

available there. Section 3.6.1 of the Users' Guide had explained already how to use this model checker.

4.3.4 Overview of the ePNK API

The previous sections have given an idea of how the ePNK and its API can be used to access and modify Petri nets for implementing some functions on Petri nets. It also showed how to plug in these functions to the ePNK – or actually to Eclipse. But, these examples just scratch the surface. In this section, we give an overview were to find and look up things in the API of the ePNK and how to use this API in the context of Eclipse and EMF. Most of these things, are actually not specific to the ePNK, but specific to Eclipse and EMF, and could be read up on in the many Eclipse publications (e. g. [2, 4, 5]). Anyway, we briefly mention or point to some of the relevant concepts here, in order to avoid some unpleasant surprises.

4.3.4.1 Eclipse and EMF

We start with giving an overview of the code that is generated[11] from Ecore models, which was also briefly discussed in Sect. 4.3.1 already. All models used in the ePNK are *Ecore models*, which are an implementation of the *Meta Object Facility* (*MOF*) [22]. For the purpose of this manual[12], an Ecore model can considered to be a simplified version of a UML class diagram. Note that, in Ecore models, the concept that represents a UML association is called *references* – and in the case of a bi-directional association, a pair of two *opposite* references.

The *Eclipse Modeling Framework* (*EMF*) [2] allows us to generate Java code from these models, which provides *getter* and *setters methods* for all the attributes and references. And there will be a lot of code behind the scenes for loading and saving models, and for notifying some observers when changes are made. Actually, in the generated code, each class of an Ecore model is represented by a Java interface and a Java class implementing this interface; the interfaces and implementations reside in two different Java packages – where typically the package name with the implementing classes ends with a segment called "impl" and also the name of the implementing Java class will end with "Impl". Normally, developers that want to access

[11] EMF provides many features to configure and change the way the code is generated from a model. Here, we discuss only the standard settings, which – with some few exceptions – are used for all models of the ePNK.

[12] We do not bother to go into the details of the MOF-levels and into the motivation behind MOF. See [15] for an brief introduction and overview.

model elements, would use the interfaces only. For attributes and references with multiplicities 1 or 1..0, the generated API and the use of the generated setter and getter methods is straightforward[13]. In case of multiplicity *, you will find that there are no setter methods for the respective attribute or reference at all; there will be a getter method, which returns a collection. In order to add or remove an element to or from the attribute or reference, you would obtain this list by the getter methods, and then add or remove something from the collection by the respective methods of collections. Note that this collection is attached to the object, and it is crucial that you do not use it for other purposes.

As explained above, the package with the Java classes that implement the Java interface of the model should typically not be used directly by other developers. This also applies to the constructor (which normally is protected). If a developer wants to create an instance of some class, this should be done via a *factory* for the model, which can be found in the same Java package as the generated Java interfaces of the model. This factory class is typically called `XXXFactory`, where `XXX` is the name of the package; the singleton instance of this class can be accessed by an attribute `eINSTANCE`. For each class of the model, this Factory provides a method for creating a new instance of the respective class.

Note that all[14] Java classes that are generated as implementations for classes from the Ecore model inherit from the class `EObject` of the EMF Framework. The class `EObject` provides a lot of functionality behind the scenes and also some convenience methods. For example, it allows another object to register with it as a listener, so that the other object is notified about any changes of its attributes and references, and even some other events. But, we do not go into these details here. One of the convenience methods is to obtain an iterator of all its directly and indirectly contained elements (indicated by compositions in the Ecore model): `eAllContents()`. All the methods of the `EObject` start with the letter "e". We cannot discuss them here; for example, there are methods for reflectively finding out which model class this object represents `eClass()`; there is a method `eContainer()` to obtain the object in which this object is directly contained; there are methods to find out which features this object has, and to change them.

Note that `EObject` has a method `eResource()`, which returns the re-

[13] There is a minor, but sometimes confusing twist when an attribute is of type boolean: in that case, the getter method actually starts with "is".

[14] Remember that we discuss the standard configuration of EMF only.

source (file) in which the object is contained – if it is associated with a file already. Resources are important, when your want to load and save models to a file, and when they are loaded and edited in an editor. Actually, a resource is typically contained in a *resource set*, which is responsible for maintaining different resources that refer to each other – and for loading and saving them together so that the links between them remain consistent. For a resource, the resource set it is contained in can be obtained by method `getResourceSet()`. In turn, resources can and should be created from a resource set, which will make sure that the correct type of resource is used for the respective file type We have seen two examples of that: in the file overview (Sect. 4.3.1), the resource set and resource was used to open a selected PNML file; in the "multi-agent mutex wizard" (Sect. 4.3.2), the resource set was used to create a new PNML file. Note that, once you have a resource, its contents can be obtained by the method `getContents()`, which returns a list of `EObjects`; which can be used to access the element, but also to add new elements. The `save()` method of the resource can be used to save the current contents of the resource to the file.

Note that the only example where we actually change (or create) a Petri net model is the "multi-agent mutex wizard" of Sect. 4.3.2. In the other examples, we access and inspect the contents of a Petri net document only. For the changes and additions we made, we could use the getter and setter methods of the API that was generated from the PNML core model. This, however, was possible only because the resource that we were changing was not under the control of an editor. If a resource is under the control of an editor, it would be under the control of a so-called *editing domain*. In that case, we cannot make changes on the resource with the getter and setter methods of the API directly anymore. Depending on which kind of editing domain it is, changes made with the API might even result in exceptions. The reason for this is that changes "on the side" by some other programme would ruin the editor's undo and redo mechanism. If a function should make changes to a model that is under the control of an editing domain, these changes need to be encapsulated into commands of the Eclipse *command framework*, which however is beyond the scope of this manual (see Sect. 3.3 of [2] for an overview of these concepts).

Eclipse provides many different ways to plug in extensions to Eclipse itself and to the ePNK. In the examples from Sect. 4.3.1–4.3.3, we used *views*, *wizards*, and *pop-up menus* for that purpose, and we used *jobs* for running long-running computations in the background. And Eclipse, provides many more possibilities, which are beyond the scope of this manual. You will find

more information on that in [4]: Chapter 6 discusses commands[15], actions and handlers; Chapter 7 discusses views and Sect. 21.8 gives a brief overview of Eclipse jobs.

For pop-up actions and handlers, the respective extension points of Eclipse allow us to provide information to which elements the respective actions and commands should apply, and when the respective actions should be visible in pop-up menus, tool-bars etc. Only when the respective element is selected these entries will be shown. This is straightforward when elements are selected in the tree editor – then the respective Java class can be used. In the graphical ePNK editor, this is slightly more tricky, since Eclipse does not "see" the underlying model elements; Eclipse "sees" only the controllers, which are called *edit parts*. If you want to attach commands and actions to the graphical editor, the actions and handlers need to be registered for these edit parts. Since the action, actually, works on the underlying model element, we need a way to access that model element from the resp. edit part, which might be a bit confusing for people new to the EMF and GMF framework. Listing 4.16 shows how to obtain a `Page` object from its corresponding edit part[16]. But, the same code would work for all other types of objects, where `Page` would need to replaced with the respective other class. Note that the method `getModel()` is actually not returning the model element behind the edit part. It returns a view of the diagram; only the `getElement()` method of this view returns the underlying model element.

4.3.4.2 ePNK models

In order to implement functions for the ePNK, you would make use of the different packages, classes, and their methods of the ePNK (in short the API of the ePNK). Since there is a standard mapping between the Ecore models and the generated API (see Sect. 4.3.4.1), we do not discuss the API explicitly; we give an overview of the models underlying the ePNK, which serve as a kind of map. The standard mapping together with the auto-completion mechanism of the Eclipse IDE, should make it possible to use the API based on these models.

[15] Note that this notion of command should not be confused with the notion of command of the EMF command framework!

[16] This code is a snippet from the action that opens a graphical editor on a page, which can be found in the project `org.pnml.tools.epnk.gmf.integration` in the class `InitiateGMFEditorOnPage` of package `org.pnml.tools.epnk.gmf.integration.actions.popup`.

Listing 4.16: Accessing the model element underlying an edit part

```
page = null;
if (selection instanceof IStructuredSelection) {
  IStructuredSelection structuredSelection =
    (IStructuredSelection) selection;
  if (structuredSelection.size() == 1) {
    Object selected = structuredSelection.getFirstElement();
    if (selected instanceof Page) {
      page = (Page) selected;
    } else if ( selected instanceof EditPart ) {
      EditPart part = (EditPart) selected;
      Object model = part.getModel();
      if (model != null && model instanceof View) {
        EObject object = ((View) model).getElement();
        if (object != null && object instanceof Page) {
          page = (Page) object;
} } } } }
```

The ePNK is based on (and generated from) many different models, most of which reside in the plug-in project `org.pnml.tools.epnk`[17] in the "model" folder. Undoubtedly, the most important model is the *PNML Core model*; it contains all the constructs common to all Petri nets (cf. Fig. 2.1 and Fig. 4.1). The PNML core model is actually split up into three diagrams, the diagram `PNMLCoreModel.ecorediag` covers the main concepts of PNML, the diagram `PNMLCoreModelGraphics.ecorediag` covers the graphical features of PNML, and `PNMLCoreModelProxies.ecorediag` covers some features that are volatile (which means that they are not saved to a file) and are responsible for maintaining the relation between the GMF diagram and the PNML information. The corresponding Java package with the interface and the factory for this package is `org.pnml.tools.epnk.pnmlcoremodel`.

Since we had discussed the main idea of the PNML core model already in Sect. 2.2.1, we do not discuss it here any further. Concerning the volatile features of `PNMLCoreModelProxies.ecorediag` and the classes `PageLabelProxy` and `LabelProxy`, we actually recommend not to use them anywhere in your functions and applications.

The model `PNMLDataTypes` defines some of the data types used in the PNML core model (note that this replaces the respective XMLDataTypes

[17] Remember that you can make the source code and the models available via the "Import As" → "Source Project" from the Eclipse "Plug-ins view" (see Sect. 4.1.1).

that are used in ISO/IEC 15909-2).

The model `PNMLStructuredPNTypeModel` provides some general infrastructure for defining more complex Petri net type definitions with labels that require some parsing and linking, which will be discussed in Sect. 4.5.3.

The other two models `PNMLPageDiagramInfo` models the GMF diagram information for the graphical editor of the ePNK, which is stored as tool specific information of the PNML model. And the model `Serialisation` represents some auxiliary information when loading some models. Both of these models, and the API generated from them are not supposed to be used for ePNK extensions. In particular, messing around with the diagram information might render graphical information inconsistent – and the graphical editor of the ePNK might not be able to start up again, when this is changed manually.

4.3.4.3 ePNK Petri net types and their use

Some of the models that come with the ePNK provide the definition of the two Petri net types of ISO/IEC 15909-2. The model for the Petri net type definition of P/T-Systems resides in the model folder of project `org.pnml.tools.epnk.pntypes`: `PTnet.ecore`. The models for the Petri net type definition of HLPNGs resides in the model folders of `org.pnml.tools.epnk.pntypes.hlpngs.datatypes` and `org.pnml.tools.epnk.pntypes.hlpng.pntd`.

Both Petri net type definitions are discussed in more detail in Sect. 4.5.

Here we point out one important aspect of using these Petri net types when adding new Petri net elements like places, transitions, and arcs to the net. Since every Petri net type can define its own kind of extensions of places, transitions and arcs, and actually also of pages, and reference nodes, it is important, that only places of that kind are used in a net of the respective kind. The tree editor as well as the graphical editor of the ePNK guarantee that always the correct type of element is created, which fits the Petri net type. The API, however, would allow to add other kinds of elements, which ultimately might result in problems when serializing and loading the net again. In order to make it easier to create the correct type of element, the class that defines a Petri net type serves as a factory for creating the respective elements: The interface `PetriNetType` which all Petri net types implement has the methods `createArc()`, `createPage()`, `createPlace()`, `createTransition()`, `createRefPlace()`, and `createRefTransition()`. And it is strongly recommended to use these methods for creating the respective elements (we have seen that in the example of Sect. 4.3.2).

Actually the interface `PetriNetType` even serves as a factory for creating the Petri net type itself and for creating a Petri net of the respective type: `createPetriNet(String)`, `createPetriNetType(String)`, and `createPetriNetType()`, where the parameter of type `String` would be the unique URI identifying the type, which is discussed in Sect. 4.5 in more detail.

4.3.4.4 ePNK convenience classes

The main purpose of the PNML core model was to define an interchange format for Petri nets. The concepts and their relation captured in the PNML core model were driven by this purpose. For actually accessing, modifying, and updating the net, the model and the API generated from it are sometimes a bit clumsy and require some extra steps in programming. In order to make up for that, the ePNK provides some convenience classes that should make some programming a bit easier. Some of the convenience classes can be found in the Java package `org.pnml.tools.epnk.helpers` in the plug-in project `org.pnml.tools.epnk`.

The first important class is `FlatAccess`, which is instantiated with some Petri net of any type. Once an instance is created, it provides methods to directly get a list of all the places and transitions. And for each node, it gives the set of all in-coming and out-going arcs. And there are two methods that, for a place or a transition, return the list of all reference places resp. reference transitions that refer to that place. And there are methods for the other direction: for a given node (which might be a reference node), the method `resolve` computes which node it actually refers to (which will be a place or a transition). This actually indicates the main purpose of the convenience class `FlatAccess`. Even though the Petri net model contains pages and places and transitions distributed among them, with `FlatAccess` it appears to be a flat net. This way, functions and applications that are interested in the Petri net only, can ignore the page structure.

An other convenience class is `NetFunctions`. It provides several static methods: e. g. there is a method that, for a given object, returns the Petri net to which this object belongs (or `null`, if it does not belong to any Petri net); there is method that returns the Petri net type of the net an object belongs to. And there are methods that return all pages of a net or all the net's objects.

The convenience classes `AbstractEPNKAction` and `AbstractEPNKJob`, make it easier to start long-running computations on some Petri net. These two classes can be found in the Java package `org.pnml.tools.epnk.`

`actions.framework.jobs` in the plug-in project `org.pnml.tools.epnk.actions`. The use of these two classes is discussed in Sect. 4.3.3.

4.4 Adding applications

In this section, we discuss the implementation of ePNK *applications*. In contrast to functions, once started, applications stay in the background, ready to interact with the user and to show results to the user (see Sect. 3.6.2 and 3.6.3 for examples). In addition, applications can visualize results by overlays on top of the Petri nets in the graphical editor and interact with the user via these overlays. The presentation of results and the interaction mechanism are still in an experimental phase. But, we will explain them in this manual anyway – if you use the presentation mechanism (and adjust it to your needs), you should be prepared to change your code in order to make them work with future versions of the ePNK.

We discuss how to implement an application by the help of the example from Sect. 3.6.2, which computes the context of every transition (the set of arcs and places directly attached to the transition), and allows the user to browse through all these transition contexts. In this example, the context of all transitions is calculated when the application is started – but this could also be done on demand, as it would be done when calculating the enabled transitions during a simulation, for example.

The implementation of the transition context application can be found in the project `org.pnml.tools.epnk.tutorials.applications`. Listing 4.17 shows the outline of the class implementing the application, where a part of the computation of the context is still missing – as indicated by ellipses. The missing part can be found in List. 4.18.

We start with the discussion of the overall structure, which is shown in Fig. 4.17. Line 1 shows that the `CalculateTransitionContext` application extends the `Application`, which is an ePNK convenience class making it easy to implement applications. It would be enough if the application implemented `IApplication`; this would, however, require much more programming. Lines 3–5 show the constructor, which does not have any exiting behaviour in its own right; since the constructor of the class `Application` expects a Petri net, this parameter is just passed on.

The actual computation of the transition context is done in the method `initializeContents()`, where for each transition, all the in-coming and out-going arcs as well as the attached places are computed, and an *object annotation* is created for each of them. The actual computation is not

Listing 4.17: Transition context application: Outline

```
public class CalculateTransitionContext extends Application {

  public CalculateTransitionContext(PetriNet petrinet) {
    super(petrinet);
  }

  public void initializeContents() {

    NetAnnotations netAnnotations = this.getNetAnnotations();
    PetriNet petrinet = this.getPetrinet();

    for (Transition transition: (new FlatAccess(petrinet)).
        getTransitions()) {
      NetAnnotation netAnnotation = NetannotationsFactory.
          eINSTANCE.createNetAnnotation();
      netAnnotation.setNet(petrinet);

      ...

      netAnnotations.getNetAnnotations().add(netAnnotation);
    }

    if (netAnnotations.getNetAnnotations().size() > 0) {
      netAnnotations.setCurrent(
          netAnnotations.getNetAnnotations().get(0));
    }
  }
}
```

yet shown – the code in lines 14–16 shows only the creating of a new *net annotation* to which the object annotations will be added later. Note that, for each net annotation, a corresponding net must be set.

After all the object annotations have been computed and added to the net annotation, the net annotation is added to the list of all the application's net annotations (which is obtained from the application by method `getAnnotation()` as shown in line 9). In the end (lines 23–25), the net annotations current annotation is set to the first one of the computed list. This will be the elements that are initially high-lighted: the context of the first transition.

Listing 4.18 shows the details of how the context is computed for each transition, and how the corresponding object annotations are created and added to the net annotation (note that the listing shows the complete for-loop again – also the part that was shown in List. 4.17 already). As said before, for each transition, a new net annotation is created (by using the respective factory of the annotation package) and the associated Petri net is set. Then an object annotation is created (by using the factory again) and added to the net annotation; for the object annotation, we need to set the object that should be annotated; in this case (line 8), it is the transition. Then, by iterating over the in-coming arcs (line 11–21), object annotations for the arcs and their source places are created and added to the net annotation. For each of these object annotations, we need to set the reference to the net object that should be annotated by it. Likewise the object annotations for the out-going arcs and the attached places are created (line 23–33). In the end, all object annotations that have been collected for the context of the transition in a single net annotation are added to the list of net annotations of this application (line 35).

This is all that needs to be done so that the transition contexts are shown as discussed in Sect. 3.6.2, once the application is started. The standard actions for the application then allow the end user to go back and forth in the list of all net annotations.

Of course, there must be a possibility for the user to start the application. In our case, this is done by a pop-up menu, which is installed on a Petri net object in the tree editor of the ePNK. This is done by standard Eclipse mechanisms and not discussed here (have a look at the class `StartApplication` and the "plugin.xml" file in project `org.pnml.tools.epnk.tutorials.applications`, if you are interested in details). It is planned for the future, to realise a mechanism to plug in ePNK applications so that they automatically show up in the ePNK menus (or a separate view).

Of course, there are other kinds of applications, where not all the an-

Listing 4.18: Transition context application: Computing the context

```
for (Transition transition: (new FlatAccess(petrinet)).
    getTransitions()) {
  NetAnnotation netAnnotation = NetannotationsFactory.
      eINSTANCE.createNetAnnotation();
  netAnnotation.setNet(petrinet);
  ObjectAnnotation objectAnnotation = NetannotationsFactory.
      eINSTANCE.createObjectAnnotation();
  objectAnnotation.setObject(transition);
  netAnnotation.getObjectAnnotations().add(objectAnnotation);

  for (Arc arc:transition.getIn()) {
    objectAnnotation = NetannotationsFactory.
        eINSTANCE.createObjectAnnotation();
    objectAnnotation.setObject(arc);
    netAnnotation.getObjectAnnotations().add(objectAnnotation);

    objectAnnotation = NetannotationsFactory.
        eINSTANCE.createObjectAnnotation();
    objectAnnotation.setObject(arc.getSource());
    netAnnotation.getObjectAnnotations().add(objectAnnotation);
  }

  for (Arc arc:transition.getOut()) {
    objectAnnotation = NetannotationsFactory.
        eINSTANCE.createObjectAnnotation();
    objectAnnotation.setObject(arc);
    netAnnotation.getObjectAnnotations().add(objectAnnotation);

    objectAnnotation = NetannotationsFactory.
        eINSTANCE.createObjectAnnotation();
    objectAnnotation.setObject(arc.getTarget());
    netAnnotation.getObjectAnnotations().add(objectAnnotation);
  }

  netAnnotations.getNetAnnotations().add(netAnnotation);
}
```

notations can be calculated in the initialisation. In that case, some of the methods of the convenience class `Application` can be overridden in order to accommodate for that. Then, it is also possible to install more or other actions than standard forward and backward buttons. In particular, overriding the `nextAnnotation()` method could be used to calculate the next annotations on demand. Note that, if an application allocates resources that need to be freed, when the application is closed, this should be done by overriding the method `shutDown()`.

Right now, all net annotations consist of a set of object annotations. Graphically, such a net annotation is always shown by a red overlay of the respective elements in the graphical editor (provided the respective page is open in a graphical editor). By some programming this behaviour can actually be changed (the simulator for high-level Petri nets of Sect. 3.6.3 is an example). But, we intend to equip applications with a *presentation description*, by which an application can define how specific annotations should be shown to the end user; e. g. by using different colours or different shapes or textual annotations on top of the graphical representation of the Petri net. Moreover, there will be an *interaction description* for applications, that will allow us to define how the end user can interact with some of the annotations by clicking on them; and which action are triggered by these user interactions.

4.5 Adding Petri net types

As mentioned several times already, it is one of the main features of the ePNK that new Petri net types can be plugged in. In this section, we discuss how to plug in a new Petri net type. In Sect. 4.5.1, we start with a simple version, for which we, basically, need to provide an Ecore model with the extensions only; as an example, we use P/T-Systems (PTNet), which come as an integral part of the ePNK; but it is defined with ePNK's type definition mechanism.

In order to explain the use of attributes in Petri nets (which do not occur in P/T-Systems and HLPNGs), Sect. 4.5.2 briefly discusses the definition of another Petri net type: signal-event nets (SE-Nets). This type is then used later again to explain how to extend the graphical representation of some features of a Petri net.

For more complex Petri net types, we can also define the mapping from the concepts of the Petri net type to their representation in XML. Some Petri net types have a quite sophisticated syntax for their textual labels, which

need to be parsed in some way – and sometimes also linked to other *symbols* of the net. Such labels are called *structured labels*, and Petri net types using such labels are called *structured Petri net types*. For these Petri net types, a *parser* and a *linker* for the structural labels must be provided. The parser is needed to convert the text of the label from its concrete syntax to its abstract syntax or "structure"; the linker is needed for linking the use of symbols in some labels to their definition in others. We use the example of high-level Petri nets (HLPNG) for discussing the relevant details in Sect. 4.5.3.

In the end, in Sect. 4.5.4, we will provide a short overview and summary of the main concepts and steps for creating a Petri net type definition

4.5.1 Simple Petri net type definitions: PTNet

The definition of P/T-Systems follows almost exactly the idea outlined already in Sect. 2.2.2, and the Ecore model that we use in the Petri net type definition is almost a copy of the one that we have seen in Fig. 2.2 on page 8 already. It remains to discuss some of the differences in these models, and to discuss the steps to make the type known to the ePNK (in short to plug it into the ePNK).

The plug-in projects that are relevant for the Petri net type definition of P/T-Systems are `org.pnml.tools.epnk.pntypes` and the mostly automatically generated project `org.pnml.tools.epnk.pntypes.edit`.

4.5.1.1 The model

The type `PTNet` is defined in project `org.pnml.tools.epnk.pntypes`. The main part is the Ecore model in "PTnet.ecore" in the folder "model", where the diagram information is contained in "PTNet.ecorediag". The diagram is shown in Fig. 4.3.

There are only some minor, but important differences, to the model from Fig. 2.2 on page 8. We discuss these differences below:

1. There is a class `PTNet` in the Ecore model, which does not occur in the conceptual model. The reason is that packages are not very tangible in programming and in the Eclipse plug-in mechanisms. Therefore, we define a Petri net type as an explicit class within that package; this class inherits from `PetriNetType` from the PNML core model (package `pnmlcoremodel`), which is not shown graphically in the diagram. It is this class (`PTNet` in our example) that is plugged in as a Petri net type definition to the ePNK later. Moreover, this class implements

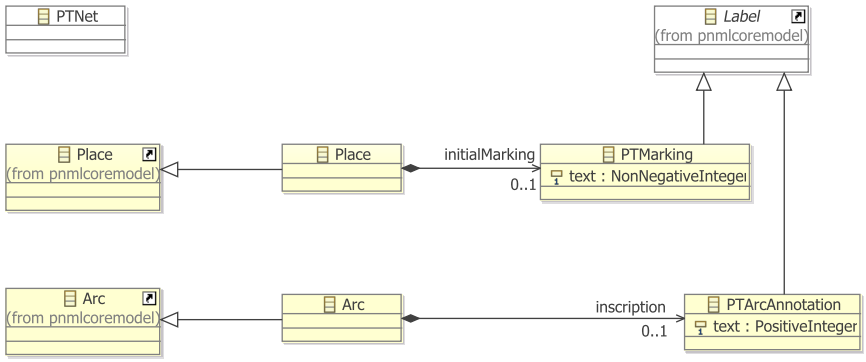

Figure 4.3: The Ecore model for the Petri net type PTNet

some methods that help the ePNK to access the information about its labels; the details, however, do not need to concern us right now.

2. There are two new classes `Place` and `Arc`, which inherit from the classes `Place` and `Arc` from the package `pnmlcoremodel`. And it is these new classes to which the additional labels are attached (initial marking and inscription). The reason for using inheritance here instead of merging packages is, that Ecore does not have the concept of merging packages[18]. Instead, the extended information for the specific Petri net type is attached to the derived classes in this new package. There could be also a class for `Page` and `Transition`, but we do not need them here, since in P/T-Systems only places and arcs have additional labels.

   Note that the name of the two classes, `Place` and `Arc` are the same as in the PNML core package, which is not ambiguous since these new classes are defined in another new package. For now, we assume that the names of these classes are the same as in the PNML core model[19].

3. The additional classes for labels, `PTMarking` and `PTAnnotation`, are

[18] It might be a good idea not to use the merge concept for extending the place and transition of the PNML core model in ISO/IEC 15909-2 when defining a new Petri net type. The merge does not work properly when nets of different types are used within the same document, but which is legal according to ISO/IEC 15909-2

[19] In principle, the names could be different; but this would require some extra programming, which we do not discuss in detail in this manual. Basically, the reflective code of the methods for creating the instance of the respective Petri net element in class `PetriNetTypeImpl` (see Sect. 4.3.4.3) need to be overridden, so that they return an object of the correct type.

attached to the new class `Place` and `Arc` as in the conceptual model via a composition – only the directive "refines" is missing, due to the missing concept of merging packages in Ecore. The features `text` are directly represented as an attribute of type `NonNegativeInteger` and `PositiveInteger`, which are predefined data types of the ePNK that represent the respective data types from XML Schema which are used in ISO/IEC 15909-2. The cardinality for the `text` attributes is 1 in both cases – the same as in the conceptual model of ISO/IEC 15909-2.

4. The new labels `PTMarking` and `PTAnnotation` are derived from the PNML core model class `Label` and not, as in the conceptual model, from `Annotation`. The reason is that the ePNK considers every label that is not an attribute to be an annotation – therefore, there is no need for an explicit class Annotation in the PNML core model of the ePNK[20].

5. A last difference is that there is no OCL constraint in this model. In the ePNK, constraints are plugged in in a different way: as EMF constraints, which is discussed in Sect. 4.5.1.4.

Such a model and diagram can be created and edited by the graphical editor of "Ecore Tools", which will not be discussed here (see the "EMF Ecore Tools Developer Guide" in the "Eclipse Help" and the web pages for some information).

Note that the Ecore package that contains the definition of a simple Petri net type must meet some conditions:

1. It must contain exactly one class that is derived from `PetriNetType`. The name of this class, however, can be chosen freely.

2. There can be classes which are derived from any of the following classes of the PNML core model: `Place`, `Transition`, `Arc`, `Page`, `RefPlace` or `RefTransition`. The names of these derived classes in the new package should the same[21].

3. These derived classes can have any number of references to some other classes. The classes that these references refer to must be derived from the class `Label` of the PNML core model, and the feature must be a

[20] Actually, there are classes `NetAnnotation` and ObjectAnnotation in the ePNK. But, these classes represent annotations on top of an existing Petri net, and are *not* annotations in the the sense of the PNML core model of ISO/IEC 15909-2 at all.

[21] By some programming, however, the names can be changed

composition (a containment feature); the name and the cardinality of the features can be chosen freely. If the feature has cardinality "many", this means that the respective object can have multiple labels of that kind.

4. The classes that are derived from class `Label` must have exactly one attribute, which has the name `text` and cardinality "1". The type of the attribute `text` can freely be chosen; it can be an Ecore built-in data type, a user-defined data type, an enumeration type, or a data type imported from other packages. The classes that are directly derived from `Label` must not have any reference[22].

5. Note that the package may also contain one class that is derived from the class `PetriNet` from the PNML core model. The name of that class can be freely chosen. The derived class may have the same kind of references as discussed for Petri net objects above. In that case, the Petri net itself can have labels attached to it[23], which are called *net labels*.

4.5.1.2 Generating the code

The code generation from that Ecore model follows the EMF standard procedure. In short, we need to generate the *model code* and the *edit code*. But, we briefly go through the process of generating all the relevant code below.

Before we can generate the code from the model, we need to create the so-called *generator model* ("genmodel"). This generator model contains some configuration information on how the code should be generated. For example, the generator model contains the information to which project and which packages, the Java code for the model should be generated. The generator model also allows us to configure the generation of the EMF tree editor; for example, we can state whether a features can be changed, whether it should be shown as a child element or as a property, etc. (see [2] for more details). The generator model can be created from an Ecore model by selecting the Ecore model ("PTnet.ecore" in our case), clicking the right mouse button, and selecting "New"→"Other..." in the pop-up menu and

[22] We will see later that this is slightly relaxed for structured labels, which are discussed in Sect. 4.5.3.2. Structured labels, however, are not directly derived from class `Label`; they are derived from class `StructuredLabel`.

[23] We would discourage to define Petri net types were labels can be attached directly to a Petri net. But since ISO/IEC 15909-2 mandates that this is possible for HLPNGs, the ePNK provides the possibility to define such net labels.

then, in the "New" dialog, choosing "EMF Generator Model" in the category "Eclipse Modeling Framework". In the case of a new Petri net type, the Ecore model has references to other models, like the PNML core model, and their "genmodels"; in the wizard for creating the "genmodel", make sure that you do not choose these other models as so-called root models, but that you add (and select) the respective generator models in the lower part as "Referenced Generator models" instead. You do not need to make any changes in the "genmodel", but we recommend that you change the "Base package" property to some reasonable path.

From the "genmodel", we can generate[24] the code for the model (*model code*), and the code with the infrastructure for all editors, which is called *edit code*. We cold also generate a simple tree editor for this model; but we do not need it; so we recommend not to generated it in order to avoid confusion and an inflation of file extensions attached to editors that are not used. Generating the code can be done by opening the "genmodel", and then selecting (after clicking the right mouse button) "Generate Model Code" and "Generate Edit Code". After that, you will find[25] the code for the model in the "src" folder of the project with the model and "genmodel". Moreover, the "plugin.xml" will make the model and its code known to Eclipse by an extension: `org.eclipse.emf.ecore.generated_package`.

The edit code is generated in a project with the same name extended by a suffix ".edit". We do not need to change anything in the edit code[26]. Note that we must generate the edit code, in order for our Petri net type definition to work properly. If we do not do that that, we will get some exceptions when using the ePNK with the new type.

4.5.1.3 Adding the Petri net type to the ePNK

After the above steps, the code for the new model is known to Eclipse. But, the ePNK will not know that there is a new Petri net type. To this end, we need to define another extension, which makes the new Petri net type known to the ePNK. Before, we can do that, we need to make two minor changes in the automatically generated code. We need to make the

[24] If you have imported the plug-in projects for P/T-Nets, the code is already generated. So, you do not need to generated anything. You would need to do that only for a new own Petri net type definition. The following discussion pretends that the P/T-Net is your new Petri net type definition were you just created the Ecore model.

[25] If you do not say otherwise in the "genmodel".

[26] In the `org.pnml.tools.epnk.pntypes.edit` project with the "edit code" for PTNets, some of the automatically generated icons in the folder `icons/obj16` have been replaced by some more appropriate images, but this is just a matter of usability.

constructor of the class that represents the new Petri net type public (by default it is protected); in our example, this concerns the constructor of the class `PTNetImpl`, which can be found in the automatically generated package `org.pnml.tols.epnk.pntypes.ptnet.impl`.

Listing 4.19 shows this class with the manually changed and extended parts marked in red. You can see the constructor, which is public now. The manual change is indicated also by the `@generated NOT` tag[27]. The second manual change is the addition of the `toString()` method. This method defines the value of the PNML type attribute for nets of that particular Petri net type: the types unique URI. Here, we use the one from ISO/IEC 15909-2 for P/T-Systems. If you define a new Petri net type, you need to make sure that you use one that is not used by other Petri net types already.

With the constructor public, we can plug in the `PTNetImpl` as a new Petri net type to the ePNK now. To this end, we use the extension point `org.pnml.tools.epnk.pntd` in the "plugin.xml". Listing 4.20 shows the relevant part from the "plugin.xml" file, which can be found in the project `org.pnml.tools.epnk.pntypes`. The attribute `point` refers to the ePNK type definition extension point, the `id` is a unique identifier for the new type within the ePNK, and the attribute `name` gives the type extension some conclusive name (we use the one from ISO/IEC 15909-2). The `type` element refers to the class that implements the new type; in our example, this is our `PTNetImpl` class – with its fully qualified name. In general, the class that is chosen here must extend the class `PetriNetTypeImpl` from the PNML core model code and which must have a public constructor (that is why we needed the manual change). The description can contain a longer description of the new net type – for P/T-Systems, we guessed that no further explanation would be needed.

If we started the runtime workbench now and used the ePNK editor, it would offer us a Petri net of the new type, when we create a child element of a Petri net document. But, it would be better to wait with that until, we have also added the constraints for connecting arcs, below.

4.5.1.4 Adding constraints

As mentioned earlier, it is not allowed in P/T-Systems to have arcs that run from places to places or from transitions to transitions or from and to pages. In the conceptual model of PTNets, this is excluded by an OCL constraint in the UML model already. In the ePNK, this constraint must be added

[27] Actually, we could just delete the tag `@generated`, but it is easier to search for and keep track of manual changes, if they are tagged with `@generated NOT`.

Listing 4.19: The class `PTNetImpl` with manual changes

```
package org.pnml.tools.epnk.pntypes.ptnet.impl;

import org.eclipse.emf.ecore.EClass;
import org.pnml.tools.epnk.pnmlcoremodel.impl.PetriNetTypeImpl;
import org.pnml.tools.epnk.pntypes.ptnet.PTNet;
import org.pnml.tools.epnk.pntypes.ptnet.PtnetPackage;


// @generated
public class PTNetImpl extends PetriNetTypeImpl implements PTNet {

  /**
   * @generated NOT
   * @author eki
   */
  public PTNetImpl() {
    super();
  }

  /**
   * @generated
   */
  @Override
  protected EClass eStaticClass() {
    return PtnetPackage.Literals.PT_NET;
  }

  // @generated NOT
  // @author eki
  @Override
  public String toString() {
    return "http://www.pnml.org/version-2009/grammar/ptnet";
  }

}
```

Listing 4.20: The extension `PTNetImpl`

```
<extension
    id="org.pnml.tools.epnk.pntypes.ptnet"
    name="PTNets"
    point="org.pnml.tools.epnk.pntd">
  <type
      class="org.pnml.tools.epnk.pntypes.ptnet.impl.PTNetImpl"
      description="Place/Transition Nets">
  </type>
</extension>
```

separately, which is done by the standard mechanisms of EMF Validation in the "plugin.xml".

Listing 4.21 shows the part of the "plugin.xml" that defines this constraint. The actual OCL constraint is defined in the bottom in the XML CDATA part (lines 31–34). This OCL expression resembles the one from the conceptual UML model, but is syntactically slightly different – which is due to the specific technology. Moreover the "headline" that states the context `Arc` is missing, since in the *EMF Validation* technology, the context is explicitly set by the `target` element, which you can find immediately above (lines 23–29); in this example, it is the `Arc` of the `ptnet` package (which we refer to by the URI that is defined in the Ecore model of the Petri net type). The declaration of the events is necessary here, since we made this constraint a *live constraint*, which means that the editors will make sure not to violate it during editing. To this end, the editor needs to know the changes of which features might violate the constraint; in our example, this is setting the source or the target of an arc.

The rest of this constraint definition is a bit more technical, and we go through it only briefly. The extension that we actually define is a *constraint provider*, which consists of the package it refers to and the constraints. In our case, the package is the PNML core model – even though it is for a specific Petri net types. The reason is that the validation always starts from the PNML core model. Constraints are defined for a category; we use the one defined by the ePNK here: `org.pnml.tools.epnk.validation`. Each constraint must have a unique `id`, must state the language it is defined in (OCL, in our example), have a `name`, a `severity`, and a `statusCode`. The status code can be freely chosen; the ePNK uses 3-digit codes starting with a 3 for constraints concerning Petri net types. The mode can be *live* or

Listing 4.21: Adding a constraint for PTNets

```
<extension point="org.eclipse.emf.validation.constraintProviders">
  <constraintProvider cache="true">
    <package
      namespaceUri="http://org.pnml.tools/epnk/pnmlcoremodel">
    </package>

    <constraints categories="org.pnml.tools.epnk.validation">
      <constraint
          id=
  "org.pnml.tools.epnk.pntypes.ptnet.validation.PT_TP_ArcsOnly"
          lang="OCL"
          mode="Live"
          name="PT or TP Arcs only"
          severity="ERROR"
          statusCode="301">
        <message>
The arc {0} must run from a place to a transition or vice versa.
        </message>
        <description>
Arcs between two places or transitions are forbidden in
P/T-nets (see Clause 5.3.1 of ISO/IEC 15909-2).
        </description>
        <target
            class="Arc:http://org.pnml.tools/epnk/types/ptnet">
          <event name="Set">
            <feature name="source"/>
            <feature name="target"/>
          </event>
        </target>
<![CDATA[
 ( self.source.oclIsKindOf(pnmlcoremodel::PlaceNode) and
   self.target.oclIsKindOf(pnmlcoremodel::TransitionNode) ) or
 ( self.source.oclIsKindOf(pnmlcoremodel::TransitionNode) and
   self.target.oclIsKindOf(pnmlcoremodel::PlaceNode) )
]]>
      </constraint>
    </constraints>
  </constraintProvider>
</extension>
```

batch; in live mode, the graphical editor will watch them and not allow edit operations that would violate them – this is what we choose in our example. Other constraints, like correctness of structured labels, might be defined to be in batch mode; then, the graphical editor will allow for syntactically incorrect labels, but the violation will be reported when explicitly validating the net. The last information in the constraint is a `message`, which is shown to the end user when the constraint is violated. The parameter `{0}` refers to the object that violates the constraint (in its String representation) – for constraints other than OCL, there could be more parameters. Moreover, there is a longer `description` of the constraint.

As mentioned above, the constraint can be formulated in different languages. It could, for example, be in Java, which would require to implement a Java class. There are some examples of Java constraints in the HLPNG definition (see Sect. 4.5.3.3). Often, Java is more convenient for implementing more complex constraints.

Note that we did not define any mapping from the concepts defined in the Ecore model of P/T-Systems to their representation in XML. The reason is that, the standard mapping is good enough: the name of the composition in which the label is contained is the XML element, and the text feature of the label is mapped to the XML element `<text>` (see Fig. 2.1 on page 10 for an example). A mapping to XML needs to be defined only when the standard mapping is not enough, or when we have structured labels, which will be discussed in Sect. 4.5.3.

4.5.2 Petri net type definitions with attributes: SE-nets

In this section, we discuss the Petri net type definition for *signal-event nets* (*SE-nets*), which we had discussed in Sect. 3.4.3 from the end user's point of view. We present this example for several reasons: First and foremost, in the definition of SE-nets, we can show how to use *attributes* in Petri net type definitions. Second, the most prominent feature of SE-net are signal arcs, which run between two transitions; and these arcs are associated with a specific graphical representation – an arc with a flash symbol. Therefore, we come back to this example later in this manual when we define the graphical appearance of Petri nets (Sect. 4.6). Third, the Ecore model for SE-nets contains one feature, which would not be legal in PNML; therefore, we can use this example to show some subtle changes in order to make this illegal feature "invisible" to PNML.

As you could see in Fig. 3.7 on page 27, signal-event nets have different kinds of arcs. Read arcs, inhibitor arcs, and signal arcs. Therefore, arcs need

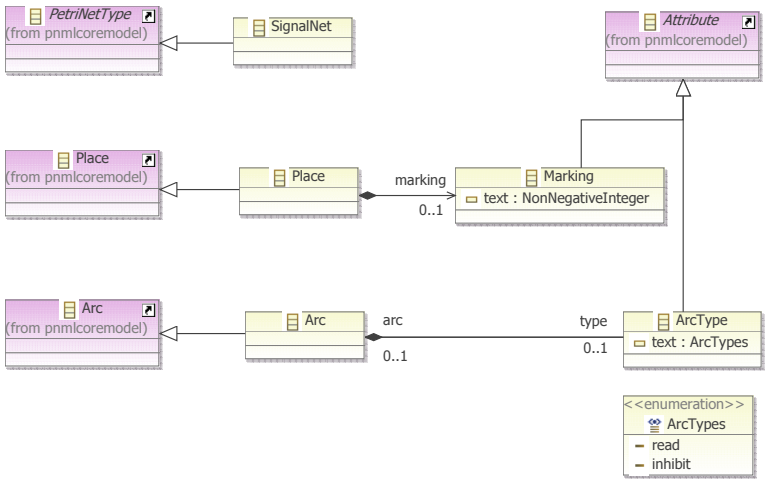

Figure 4.4: The Ecore model for SE-Nets

a label that indicates that type. The type definition for SE-nets is defined in the ePNK plug-in projects `org.pnml.tools.epnk.pntypes.signalnets` and `org.pnml.tools.epnk.pntypes.signalnets.edit`[28].

Figure 4.4 shows the Ecore model with the Petri net type definition for signal-event nets, which can be found in the folder "model" of plug-in project `org.pnml.tools.epnk.pntypes.signalnets`. In this model, the class `Arc` is equipped with an `ArcType`, which has a `text` attribute, the type of which is the enumeration `ArcTypes` – also defined in this model. Note that the enumeration has only two possible values `read` for read arcs and `inhibit` for inhibitor arcs. If there is no `ArcType` the arc is considered to be a normal arc, when it is running between a place and transition or vice versa, or as a signal arc, when it is running between two transitions. Moreover, the Ecore model defines that there can be a `Marking` for places. Both label classes inherit from `Attribute`, which means that these labels are not shown as annotations, but in the properties view, when the respective arc or place are selected in the editor (see Fig. 3.7). We will see Sect. 4.6, how the value of these attributes can be shown in the graphical representation of the place or the arc. All we need to do for a label in the Petri net type definition to be

[28] Unfortunetely, these two rojects were configured in such a way that you cannot import the source code of these projects as discussed earlier. Therefore, the source code is made available separately – you can download the source code of the respective projects from `http://www2.imm.dtu.dk/~ekki/projects/ePNK/install-details.html`; and then import them to the workspace.

considered an attribute by the ePNK is deriving it from the class `Attribute` of the PNML core model.

If we have a closer look at the Ecore model from Fig. 4.4, we see that the class `ArcType` has a reference `arc` which points back to the arc that "owns" that type – the reference `arc` is, actually, an opposite of reference `type`. This additional reference allows us to navigate back from the arc type to the respective arc, which makes it easier to formulate some constraints for arcs and their arc types[29]. As we had discussed earlier in Sect. 4.5.1.1, classes derived from `Label` and also from `Attribute` are not allowed to have any feature other than the `text` attribute. The reason for this restriction is that PNML does not allow us to serialize this feature. So, we need to make sure that such references to arcs are not serialized – conceptually this is not necessary anyway, since the reference `arc` is the opposite of the reference `type`. Therefore, we switch the serialisation of the feature `arc` off. This can done by selecting the resp. reference in the editor for the Ecore model, and then, in the properties view, selecting the "Advanced"[30] section, and then set the property "transient" to "true", meaning that this feature is not serialised to a file (the ePNK serialisation mechanism takes that into account).

From the Ecore model above, we can create the "genmodel" and generated the model code and the edit code as discussed in Sect. 4.5.1.2. And we would need to do the same manual change: implement the `toString()` method, so that it returns the unique URI for that type; and we would need to make the constructor of the class `SignalNetImpl` public. In this example, however, we chose a different way – we create a class `SignalNetFactory` that inherits from `SignalNetImpl` without any additional attributes, methods or constructors. Since the implicit default constructor of `SignalNetFactory` is public, this will do the job. This is actually the preferred method, since regeneration of the model code after a model change does not need any manual changes anymore.

Plugging in the Petri net type to the ePNK extension point works as described in Sect. 4.5.1.3. Listing 4.22 shows the resulting fragment of the "plugin.xml" (with some minor omissions).

Listing 4.23 shows the constraint for SE-nets, which makes sure that an arc type can only be present for arcs that run from a place to a transition; it also guarantees that arcs run from a place to a transition, from a transition

[29] In the current version, this feature is not used, though.

[30] This "Advanced" section shows all kinds of advanced setting of the respective Ecore element. In case you are new to Ecore and you do not understand the concept of "transient", do not worry. Just ignore this for now.

Listing 4.22: Plugging in SE-Nets

```
<extension
    id="org.pnml.tools.epnk.pntypes.signalnets"
    name="Signal Nets"
    point="org.pnml.tools.epnk.pntd">
  <type
      class="org.pnml. ... .signalnets.factories.SignalNetFactory"
      description="Signal nets">
  </type>
</extension>
```

to a place, or between two transitions. It is a live constraint, which needs to be checked, whenever the source or target of an arc are set, and whenever the arc type is set.

With these definitions, the ePNK would know what SE-nets are – still the inhibitor arcs and the signal arcs would not yet appear as shown in Fig. 3.7. To this end, we still need to extend the graphical appearance of SE-nets, which will be discussed in Sect. 4.6.

4.5.3 Petri net type definitions in general: HLPNG

In this section, we discuss some more advanced mechanisms that can be used for defining new Petri net types. These mechanism will be discussed by the help of the Petri net type definition of high-level Petri nets (HLPNGs). Therefore, we start with an overview of the concepts of HLPNGs, from the implementation point of view (for the conceptual part we refer to Sect. 3.5.2 and for a detailed discussion of all models and concepts, we refer to [6]).

4.5.3.1 Overview of HLPNGs

As discussed in Sect. 3.5.2, HLPNGs have different kinds of complex labels: *declarations* of variables, sorts, and operators; *types* defining the sort of the tokens of a place, *markings* which are multiset terms defining the initial marking of a place, *conditions* as transition guards, and *arc annotations* that define which tokens are consumed, resp. produced when a transition fires. What is more, the labels cannot be considered isolated from each other anymore – some labels, like markings, arc annotations, or conditions may use *symbols* that are defined in other labels – in particular, in the declarations.

Figure 4.5 shows the Ecore model defining the concepts of HLPNGs,

Listing 4.23: Adding the constraint for SE-nets

```
<extension point="org.eclipse.emf.validation.constraintProviders">
  <constraintProvider cache="true" mode="Live">
    <package
        namespaceUri="http://org.pnml.tools/epnk/types/signalnets">
    </package>
    <constraints categories="org.pnml.tools.epnk.validation">
      <constraint
          id="org.pnml. ... .validation.correct-arc-connection"
          lang="OCL" mode="Live"
          name="Arc connection constraint for signal nets"
          severity="ERROR" statusCode="401">
        <message>
          The arc {0} with this arc type is not allowed ...
        </message>
        <description>
          Arcs must be between a place and a transition, ...
        </description>
        <target
            class="Arc:http://org.pnml.tools/epnk/types/signalnets">
          <event name="Set">
            <feature name="source"></feature>
            <feature name="target"></feature>
            <feature name="type"></feature>
          </event>
        </target>
        <![CDATA[
        ( self.source.oclIsKindOf(pnmlcoremodel::PlaceNode) and
          self.target.oclIsKindOf(pnmlcoremodel::TransitionNode) )
        or
        ( self.source.oclIsKindOf(pnmlcoremodel::TransitionNode) and
          self.target.oclIsKindOf(pnmlcoremodel::PlaceNode) and
          self.type->size() = 0 )
        or
        ( self.source.oclIsKindOf(pnmlcoremodel::TransitionNode) and
          self.target.oclIsKindOf(pnmlcoremodel::TransitionNode) and
          self.type->size() = 0 )
        ]]>
      </constraint>
    </constraints>
  </constraintProvider>
</extension>
```

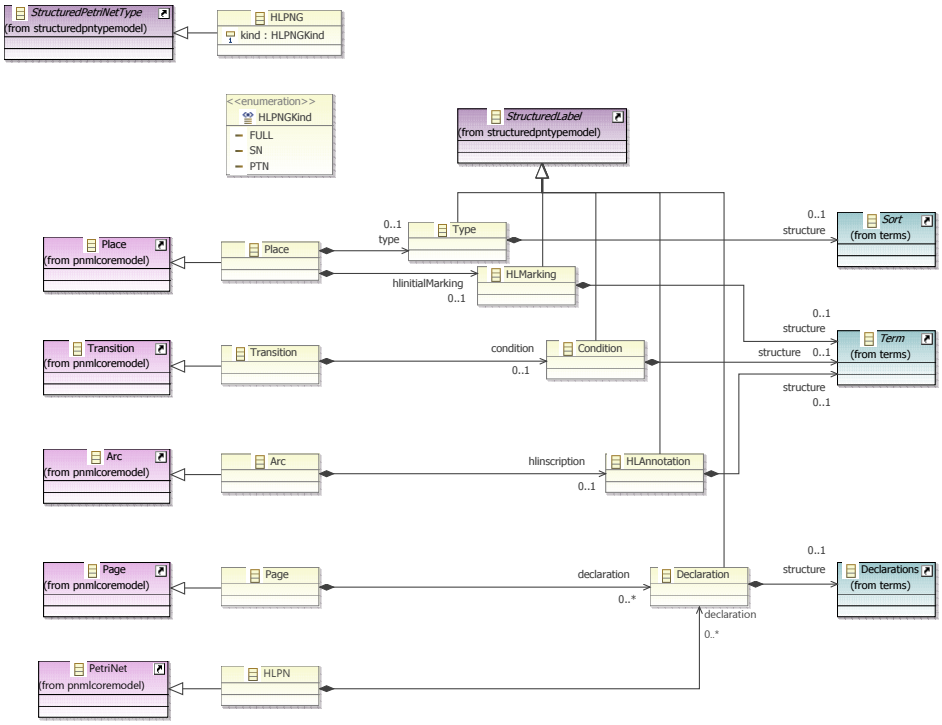

Figure 4.5: The Ecore model for HLPNGs

which can be found in the folder "model" in project[31] `org.pnml.tools.epnk.pntypes.hlpngs.pntd`. This model follows the same principles as the model for PTNets, which was discussed in Sect. 4.5.1.1. The main differences are that the defined Petri net type `HLPNG` extends a more advanced class `StructuredPetriNetType`, and all labels extend `StructuredLabel`, which are part of the PNML core model. These two classes provide the infrastructure needed for parsing the textual labels and for establishing the links between these labels. This structure is discussed in Sect. 4.5.3.2.

The actual contents of all these labels is defined in their containment `structure`; note that we use `Term` as the contents for the labels `HLMarking`, `Condition`, and `HLAnnotation`, since all of them are terms – just with differ-

[31] This is the plug-in in which HLPNGs are plugged into the ePNK; since HLPNGs are quite complex, and require many models, and also the implementation of a parser, the underlying concepts are defined in different other projects; all of these projects have a name with prefix `org.pnml.tools.epnk.pntypes.hlpngs` – some of them are generated automatically from models or from a grammar. You can import all of these projects to your workspace by the Eclipse "Import As" feature in the "Plug-ins" view.

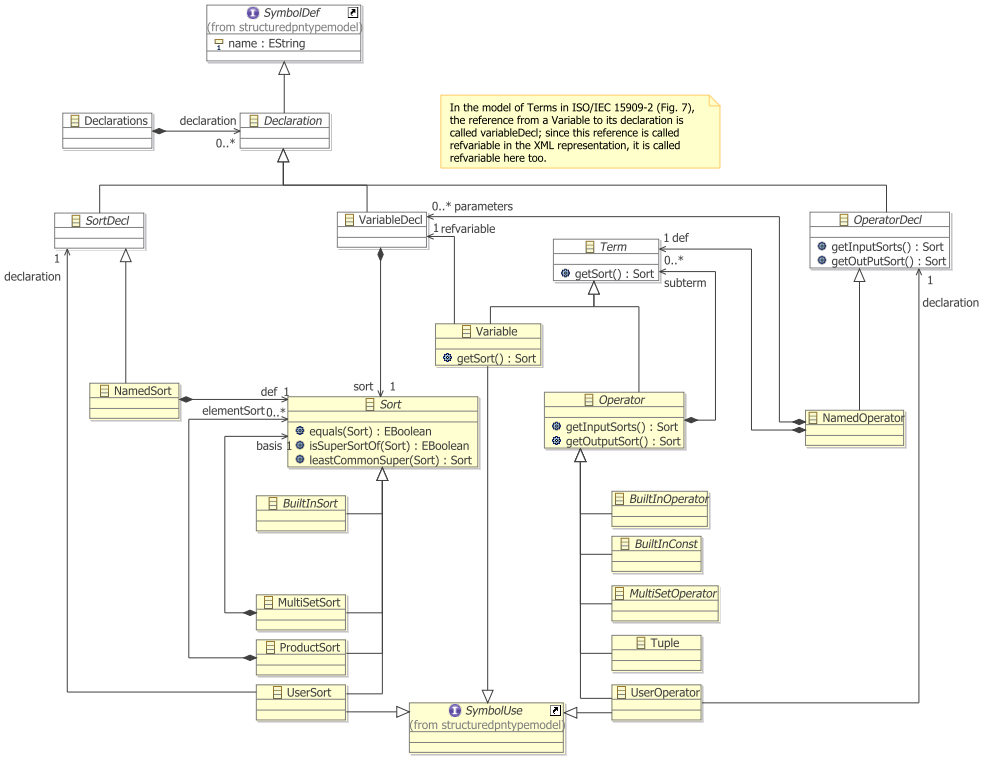

Figure 4.6: The Ecore model for the main concepts of HLPNGs

ent additional constraints imposed on them (see Sect. 4.5.3.3). Note that by contrast to normal labels and attributes, *structured labels* can have – actually must have – a composition, which normally[32] has the name `structure`. But there should not be any other features than that.

The detailed structure and concepts of terms, sorts, and declarations, are defined in several other models. Since these details are not too relevant for understanding the definition of structured Petri net types, we discuss only the main part of that model. This part of the model is shown in Fig. 4.6 – this as well as the diagrams of all the other models can be found in the plug-in `org.pnml.tools.epnk.pntypes.hlpngs.datatypes`. For a detailed discussion of these models and their concepts, we refer to [6]. There is only one important difference, which are the classes `SymbolDef` and `SymbolUse`, which do not occur in the models of ISO/IEC 15909-2. These two classes

[32]The name could be changed, but this would require some programming, which will be discussed later.

are the ePNKs infrastructure for dealing with the definition of symbols and their use in a uniform and generic way – on the side, making the concepts of *symbol definition* and *symbol use* explicit, so that the model is more concise. These concepts are part of the PNML core model concerning structured Petri net types, which will be discussed in the next section.

One other issue worth noting in the Ecore model of Fig. 4.5 is the class `HLPN`, which extends class `PetriNet` from the PNML core model. This represents the Petri net itself. Normally, Ecore models defining a new Petri net type would not need to extend the class `PetriNet` itself; it would be enough to extend the class `PetriNetType`. HLPNGs, however, have so-called *net labels*, which are labels that are directly attached to the net – and not to a page. For net types with net labels, the class `PetriNet` must be extended and equipped with compositions to the respective labels – in our example, these are declarations. But, we would discourage defining such net labels for Petri nets types.

4.5.3.2 Structured Petri net types and structured labels

As mentioned above, the ePNK provides some general interfaces and infrastructure for defining structured Petri net types, which distill the general concepts of more complex Petri net types. This is, again, captured in models (and the code generated from them).

The model for structured Petri net types can be found in the "model" folder of the ePNK core project `org.pnml.tools.epnk`: `PNMLStructuredPNTypeModel`. The diagram is shown in Fig. 4.7. We know the classes `PetriNetType` and `Label` as well as the interface `ID`, which is used for all ePNK elements that have an id, already from the PNML core model. The abstract class `StructuredLabel` extends the class `Label`, it has an attribute `text`, which stores the contents of this label as a text String. The actual structural contents is defined by classes that extend it (we have seen some examples in Fig. 4.5 already). Since, the ePNK cannot not know these concrete implementations, classes extending the structural label must make the reference to this structural contents known to the ePNK. This is achieved by the method `getStructuralFeature()`; as long as the feature for the structure is called 'structure' in the model, we do not need to do anything in the implementation (the ePNK will access this feature in a reflective way); only if for some reason, the model chooses a different name, this method must be implemented manually. Moreover, every class for a structural label must provide a method `parse()` for parsing a String – a representation of this label in concrete syntax; an implementation of this method may return

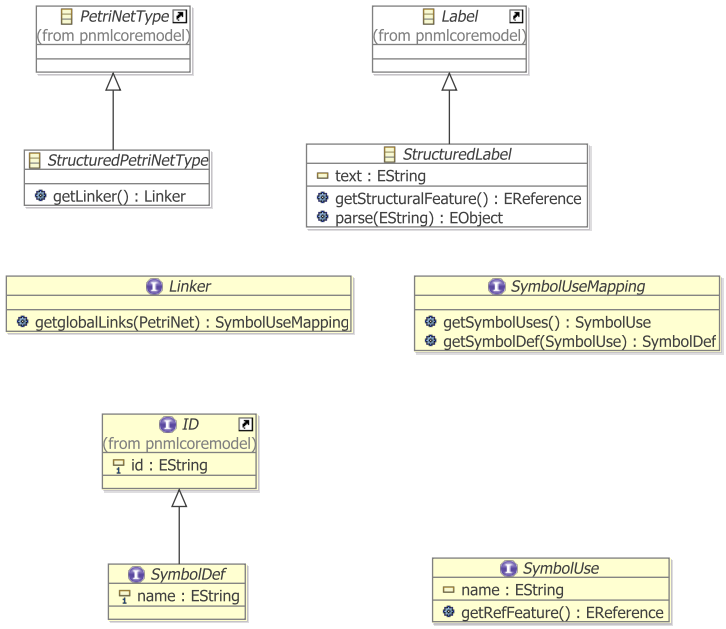

Figure 4.7: The model for structured Petri net types

null, if the text cannot be parsed. If the label could be parsed, it must return some object (to be precise an `EObject` which is the EMF version of objects) with all the substructure of that label – the abstract syntax of the label. In particular, that object must have a type that is compatible with the label's structural feature. This method must be implemented manually for every new extension since the ePNK cannot guess the concrete syntax.

The abstract class `StructuredPetriNetType` has one additional method, which must provide a `Linker` for linking the uses of some symbols to their definitions, which are captured by classes `SymbolDef` and `SymbolUse`. A `SymbolDef` has an `ID` and has a name, which will be used to refer to it (the id is internal to PNML and the ePNK). This name will be used in `SymbolUse`, again as attribute name, to refer to the definition. The feature that actually refers to the definition, can be accessed via the method `getRefFeature()`. Since the ePNK does not know anything about how to make these connections, the Petri net type needs to provide access to the linker; to this end, the class `StructuredPetriNetType` has a method `getLinker()`, which must be implemented by classes that extend it. `Linker` is an interface: a single method `getglobalLinks()`, which takes a Petri net

and returns a `SymbolUseMapping`, which is also an interface. Conceptually, the class `SymbolUseMapping` maps every `SymbolUse` to its definition `SymbolDef`. All the symbol uses for which there exists a mapping, can be obtained (as a list) via the method `getSymbolUses()`; and for each symbol use, the method `getSymbolDef()` will return the definition of that symbol.

With this infrastructure, the ePNK can deal with all kinds of structured labels. We will have a look at the implementation of some examples next: We consider the label `Condition` in the Petri net type definition for HLPNGs again (see Fig. 4.5) – the other labels are similar. Its structural feature is the containment `structure` to class `Term`. Since this is the standard name for structured labels, we do not need to override the method `getStructuralFeature`. But, we need to implement the `parse()` method. The parsers for all labels of HLPNGs were automatically generated by Xtext, and are made available in a singleton class `HLPNGParser` in package `org.pnml.tools.epnk.pntypes.hlpngs.datatypes.concretesyntax` in a project with the same name. For parsing a term, class `HLPNGParser` provides a method `parseTerm(String)`. This singleton and its method `parseTerm()` is used in the implementation of `ConditionImpl` (you will find it in the package `org.pnml.tools.epnk.pntypes.hlpng.pntd.hlpngdefinition.impl` in project `org.pnml.tools.epnk.pntypes.hlpng.pntd`).

Since linking is across all the different labels of a net, there is only a single linker for every net. For HLPNGs, this is implemented in the package `org.pnml.tools.epnk.pntypes.hlpngs.datatypes.concretesyntax.linking` in project `org.pnml.tools.epnk.pntypes.hlpngs.datatypes.concretesyntax`. This class is `HLPNGLinker`; basically it goes through the complete Petri net twice; in the first round, it creates a symbol table of all symbol definitions; in the second round, this symbol table is used to look up the definition for every symbol use, which is stored in the `SymbolMapping`, which implements the `SymbolUseMapping` that we discussed above.

To make this linker known to the ePNK, the class `HLPNGImpl` in package `org.pnml.tools.epnk.pntypes.hlpng.pntd.hlpngdefinition.impl` implements the method `getLinker()`: it returns an instance of `HLPNGLinker`.

Note that, in order to plug in the Petri net type definition to the ePNK, we need to make the constructor public in the class `HLPNGImpl`, and we need to implement the `toString()` method so that it returns the unique URI of HLPNGs method (as discussed in Sect. 4.5.1.3).

4.5.3.3 Constraints

For HLPNGs, we needed to implement quite many constraints. As an example for a *Java constraint*, we discuss one of these constraints here. The rest of them would not provide much insight into the mechanisms of the ePNK – though they might provide some insights to the inner workings of HLPNGs themselves. There is also an OCL constraint that forbids connecting places with places and transitions with transitions. But, this is exactly the same as for PTNets, which is why we do not discuss it here again.

All constraints for HLPNGs are defined in the project `org.pnml.tools.epnk.pntypes.hlpng.pntd`, the implementations of the Java constraints can be found in the package `org.pnml.tools.epnk.pntypes.hlpng.pntd.validation` We discuss the constraint that transition conditions must have type boolean, which is implemented in class `ConditionIsBoolType`. Listing 4.24 shows this class. This constraint extends the class `AbstractModelConstraint` from EMF Validation and implements the method `validate()`. From the validation context, it obtains the target object, which should be a transition (see later). But, we are defensive and check that explicitly. Then, we obtain the condition label of that transition, and if it is not `null`, get the term (its structure). Then, we check whether the sort of the term is boolean[33]. If it is not, we return a failure status via the validation context, and add the transition and the textual label to an array of objects (which is used in the error message to be defined later). Otherwise, we return a success status. Note that the EMF Validation Framework makes sure that this validate method is called for all transitions of a selected Petri net, Petri net document or page, once it is properly plugged in, which is discuss below.

Plugging in a Java constraint is similar to plugging in OCL constraints. The relevant fragment of the "plugin.xml" is shown in Listing 4.25. The main differences are that the attribute `lang` is "Java" now and the attribute `class` refers to the class `ConditionIsBoolType`, which was discussed above. As target class, the transition class of HLPNGs is defined (that is why we could assume that the target object is a transition). Another difference is that this is no live constraint, but a batch constrain. This means, that the constraint might be violated during editing; a violation will be detected and reported only when the user explicitly invokes the validation. Since this is a batch constraint, we do not need to declare any events in the target.

Another difference to the OCL constraint is, that we can refer to several

[33]The implementation of `getSort()` for terms is actually quite complex; it amounts to implementing a type system for the annotation language of HLPNGs. But we do not discuss the details here.

Listing 4.24: The constraint that conditions have type boolean

```
package org.pnml.tools.epnk.pntypes.hlpng.pntd.validation;

import org.eclipse.core.runtime.IStatus;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.validation.AbstractModelConstraint;
import org.eclipse.emf.validation.IValidationContext;
import
  org.pnml.tools.epnk.pntypes.hlpng.pntd.hlpngdefinition.Condition;
import
  org.pnml.tools.epnk.pntypes.hlpng.pntd.hlpngdefinition.Transition;
import
  org.pnml.tools.epnk.pntypes.hlpngs.datatypes.booleans.Bool;
import org.pnml.tools.epnk.pntypes.hlpngs.datatypes.terms.Sort;
import org.pnml.tools.epnk.pntypes.hlpngs.datatypes.terms.Term;

public class ConditionIsBoolType extends AbstractModelConstraint {

  public IStatus validate(IValidationContext ctx) {
    EObject object = ctx.getTarget();

    if (object instanceof Transition) {
      Transition transition = (Transition) object;
      Condition condition = transition.getCondition();
      if (condition != null) {
        Term term = condition.getStructure();
        if (term != null) {
          Sort sort = term.getSort();
          if (sort != null) {
            if (!(sort instanceof Bool)) {
              return ctx.createFailureStatus(
                  new Object[] {transition,
                      condition.getText()});
            }
          }
        }
      }
    }
    return ctx.createSuccessStatus();
  }
}
```

Listing 4.25: Adding the constraint for conditions

```
<extension point="org.eclipse.emf.validation.constraintProviders">
  <constraintProvider cache="true">
    <package
      namespaceUri="http://org.pnml.tools/epnk/pnmlcoremodel">
    </package>

    <constraints categories="org.pnml.tools.epnk.validation">
    ...
      <constraint
        lang="Java"
        class="org.pnml. ... .validation.ConditionIsBoolType"
        severity="ERROR"
        mode="Batch"
        name="Condition is of type boolean"
        id="org.pnml. ... .validation.ConditionIsBoolType"
        statusCode="314">
        <target class=
  "Transition:http://org.pnml.tools/epnk/pntypes/hlpng/pntd/hlpng"/>
        <description>
  The condition must be of type BOOL.
        </description>
        <message>
  The condition {1} of transition {0} is not of type BOOL.
        </message>
      </constraint>
    ...
    </constraints>
  </constraintProvider>
 </extension>
```

parameters in the message now. What the different parameters are, depends on the return value of the validation method. In our case, this was the transition (or its String representation) and the text of the label.

The ellipses ("...") indicate that the constraint that we have discussed here, is just one of many other constraint, which are not discussed here.

4.5.3.4 XML Mappings

In the sections above, we have discussed how to define a Petri net type and all its concepts and constraints. For saving it in PNML, it is also necessary to define how these concepts are represented in XML – at least if the "standard mappings" do not work.

In this section, we discuss how these mappings are defined. Conceptually, these mappings are tables (in ISO/IEC 15909-2, these tables are given in Clause 7.3.1). In the ePNK, these tables are "programmed" as part of the new Petri net type[34].

We explain the concepts of these "programmed tables" by discussing some of the mappings for HLPNGs. The tables for a new Petri net type are programmed, by overwriting the method `registerExtendedPNMLMetaData( ExtendedPNMLMetaData metadata)` of `PetriNetType`; the parameter `meta data` represents the table, to which the entries should be added when the method is called.

Let us have a look at some examples. Listing 4.26 shows an excerpt of the `registerExtendedPNMLMetaData()` method of the class `HLPNGImpl`, which implements the Petri net type for HLPNGs. Each of the `metadata.add` statements defines one table entry, which defines the mapping of one specific feature of the Ecore model to an XML element (we will see later how to map an Ecore attribute to an XML attribute). The three statements shown in Listing 4.26 define how the structure feature of the labels `Type`, the `HLMarking`, and the `Condition` are mapped to the XML element `<structure>`. We discuss the first one, the `Type`, in more detail:

- The first parameter, denotes the feature that is mapped to XML by this entry; in this case, it is the composition from the class `Type` to the class `Sort` (see Fig. 4.5 on page 111). The source and target classes are mentioned explicitly as second and third parameter again. We refer to the feature and the two classes via the singleton classes that describes

[34]It might be, that a future version of the ePNK will provide a means to plug in these tables directly in some form; but since "programming the tables" is not too difficult, this does not have a high priority.

Listing 4.26: Mappings for type, marking, and condition extensions

```
public void registerExtendedPNMLMetaData(
    ExtendedPNMLMetaData metadata) {
    ...

  metadata.add(
      HlpngdefinitionPackage.eINSTANCE.getType_Structure(),
      HlpngdefinitionPackage.eINSTANCE.getType(),
      TermsPackage.eINSTANCE.getSort(),
      "structure",
      null,
      HLPNGFactory.getHLPNGFactory());

  metadata.add(
      HlpngdefinitionPackage.eINSTANCE.getHLMarking_Structure(),
      HlpngdefinitionPackage.eINSTANCE.getHLMarking(),
      TermsPackage.eINSTANCE.getTerm(),
      "structure",
      null,
      HLPNGFactory.getHLPNGFactory());

  metadata.add(
      HlpngdefinitionPackage.eINSTANCE.getCondition_Structure(),
      HlpngdefinitionPackage.eINSTANCE.getCondition(),
      TermsPackage.eINSTANCE.getTerm(),
      "structure",
      null,
      HLPNGFactory.getHLPNGFactory());

    ...
}
```

the elements of the packages (HLPNGdefinition and Terms), which are automatically generated by EMF. These *package classes*, provide access to all the classes and features within a package (see [2] for more details). Note that `HlpngdefinitionPackage.eINSTANCE` refers to the package `hlpngdefinition` and `TermsPackage.eINSTANCE` to the package `terms`.

- As mentioned above, the second parameter denotes the class to which the feature belongs (it could be a sub-class of `Type` in principle); this is often called the *container class*.
- The third parameter denotes the class that the feature refers to (this could also be a sub class of `Sort`); this is often called the *object class*.
- The forth parameter defines the XML representation, the string that will be used as XML element in the serialisation of this feature (in our example "structure").
- The fifth parameter could refer to an XML attribute, that might be necessary for creating an Ecore object from the XML element (we will discuss an example later). In most cases, this XML attribute is not needed, since the XML element (and the context in which it occurs) provide enough information for creating the Ecore element from it.
- The last parameter refers to a factory that is capable of creating an Ecore instance of the respective class from the XML element and – if provided – the XML attribute. This parameter can be left empty, when the Ecore instance can be constructed reflectively from the information on the object class only.

The ePNK uses this table and its entries in two directions: In the one direction, the table is used to serialise a Petri net to its XML syntax; in the other direction, the table is used to create the model elements from the XML syntax. In the latter case, the *factories* play an important role. Listing 4.27 shows the interface that all these factories must implement. The methods `canCreateObject()` and `createObject()` have the same parameters, which basically reflect the entries of the table that we discussed above. Only the third (representing the object class) and the six one (the factory itself) are missing. And, there is an additional parameter (`provider`), which will provide access to the values of all attributes of the currently read XML element (in case the factory needs the values of some of the XML element's attributes for creating an object of the appropriate type). The method

Listing 4.27: Interface `Factory`

```
package org.pnml.tools.epnk.pnmlcoremodel.serialisation;

import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.EStructuralFeature;

public interface IPNMLFactory {

  public boolean canCreateObject(
      EStructuralFeature feature,
      Object container,
      String element,
      String attribute,
      IAttributeProvider provider);

  public EObject createObject(
      EStructuralFeature feature,
      Object container,
      String element,
      String attribute,
      IAttributeProvider provider);

  public Object createAttributeObject(
      Object object,
      String attribute,
      IAttributeProvider provider);

}
```

Listing 4.28: Mappings of an attribute

```
metadata.addAttributeMapping(
    BooleansPackage.eINSTANCE.getBooleanConstant_Value(),
    BooleansPackage.eINSTANCE.getBooleanConstant(),
    "value",
    HLPNGFactory.getHLPNGFactory());
```

`canCreateObject()` is used to find out whether the factory is able to create an object from the provided information, the `createObject()` method is used to actually create it. The `createAttributeObject` is used to create an object for some XML attribute. The implementation of these factories is straightforward and a bit boring – we do not discuss the details here. You can have a look into the class `HLPNGFactory` in package `org.pnml.tools.epnk.pntypes.hlpng.pntd.hlpngserialisation.factory` in the project `org.pnml.tools.epnk.pntypes.hlpng.pntd` to get some inspiration. What is more, with an extension that came into version 0.9.0 of the ePNK, the factory can be set to `null`. In which case the standard mechanism for creating an object of the target class will be used; therefore, we need factories only in very special cases. In most of the cases, the factory can be set to `null`[35].

Listing 4.28 shows an example[36] of how a feature of the model can be mapped to an XML attribute. In this example, the value of the boolean constant is mapped to the XML attribute `value`. This is where the method `createAttributeObject()` of the factory comes into play.

The discussion above, gives a general idea of how these tables and mappings work. All this, however, could have been achieved with the existing mechanisms of EMF: Extended Metadata. Some of the PNML constructs cannot be mapped to XML by the mechanisms provided by EMF Extended Metadata. Therefore, the ePNK needed to provide its own mechanism for mapping Ecore concepts to XML. In the rest of this section, we discuss some of these special situations.

To this end, we consider the serialisation of the simple term `x‘f(x,x)`, where `x` is a variable and `f` is a user defined operator. The PNML repre-

[35]Note that except for two features, which were used to test this new mechanism, the mappings for HLPNGs have not been updated yet; therefore, you will find factories all over these mappings. But, this has historic reasons only and will eventually be changed (making the mappings more maintainable and easier to understand).

[36]Actually, this is the only example of this kind in HLPNGs.

Listing 4.29: PNML representation of `x‘f(x,x)`

```
<numberof>
  <subterm>
    <variable refvariable="5"/>
  </subterm>
  <subterm>
    <useroperator declaration="1">
      <subterm>
        <variable refvariable="5"/>
      </subterm>
      <subterm>
        <variable refvariable="5"/>
      </subterm>
    </useroperator>
  </subterm>
</numberof>
```

sentation is shown in Listing 4.29, where "5" is the unique id of variable `x` and "1" is the id of the user defined operator `f`, In addition to being a bit verbose, there is one thing that is special about this mapping: There is an XML element `<subterm>` for the association form the top-level term (number of) to its subterm, which are represented as two other XML elements, `<variable>` and `<useroperator>`. The XML element `<subterm>` defines to which feature of the term the XML element that is contained in it should go. The XML element inside (e. g. `<variable>`) defines the type that this object should have.

The problem here, is that there is an intermediate XML element that has no object as counter part in the model – it represents an association. We call such an XML element an *association element*. The mapping for these association elements is shown in Listing 4.30. The first entry is actually as we have seen it before. The only difference is that the factory produces an instance of a new class `TermAssoc`, which has the nature of a term but, actually, represents an association to a term. We will discuss that class in more detail later. The two other mappings, define the mapping of variables and user operators to XML, and these are different, since they do not refer to any feature at all. They just refer to a container class and a contained class. The container class is the class `TermAssoc`, which will make sure that the variable resp. user operator will be added to the subterm feature of the

Listing 4.30: Mappings of associations to XML elements

```
metadata.add(TermsPackage.eINSTANCE.getOperator_Subterm(),
    TermsPackage.eINSTANCE.getOperator(),
    TermsPackage.eINSTANCE.getTerm(),
    "subterm",
    null,
    HLPNGFactory.getHLPNGFactory());

metadata.add(null,
    HlpngserialisationPackage.eINSTANCE.getTermAssoc(),
    TermsPackage.eINSTANCE.getVariable(),
    "variable",
    null,
    HLPNGFactory.getHLPNGFactory());

metadata.add(null,
    HlpngserialisationPackage.eINSTANCE.getTermAssoc(),
    TermsPackage.eINSTANCE.getUserOperator(),
    "useroperator",
    null,
    HLPNGFactory.getHLPNGFactory());
```

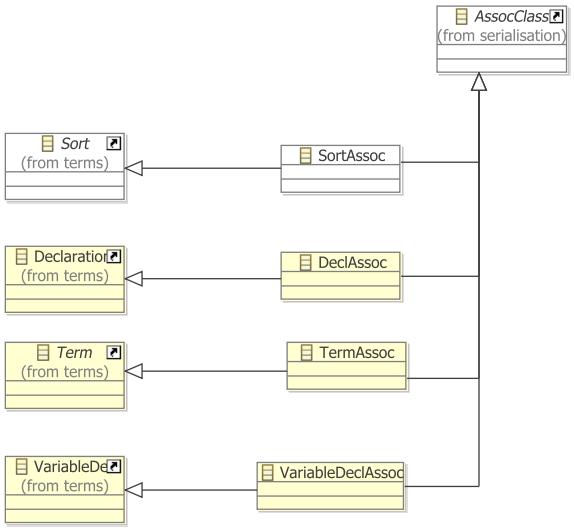

Figure 4.8: The package `hlpngserialisation`

operator on the level above[37].

The class `TermAssoc` does not need to be programmed. This class, as well as the other classes for representing association elements, could completely be generated from a model. This model is shown in Fig. 4.8. These classes extend a specific class of our model (the one to which the respective association should go), and the general class for `AssocClass`, which is defined by the ePNK, and implements all the necessary functionality. Note that these classes will not occur in the model anymore, once it is completely loaded – they are only used while a PNML file is loaded.

In the case of subterms, every subterm occurs in a separate `<subterm>` element – even if a term has several subterms, there is one subterm element for each of them (see Listing 4.29). In the case of parameters of an operation declaration, this is different: Listing 4.31 shows the PNML representation of the declaration of a named operator `f(x:INT, y:INT) = x * y`. Here, all variable declarations occur in the same `<parameter>` element. We called these *bundled association elements*. The table entries for this mapping are shown in Listing 4.32. The first one, is almost the same as for association elements, and the Factory `HLPNGFactory` would create an instance of `VariableDeclAssoc` for an XML element `<parameter>`. The new last pa-

[37]There would actually be another way of doing this, in a slightly more elegant way when using "standard features", which will be discussed later in this section.

Listing 4.31: PNML structure for declaration `f(x:INT, y:INT) = x * y`

```
<namedoperator id="1" name="f">
  <parameter>
    <variabledecl id="2" name="x">
      <integer/>
    </variabledecl>
    <variabledecl id="3" name="y">
      <integer/>
    </variabledecl>
  </parameter>
  <def>
    <mult>
      <subterm>
        <variable refvariable="2"/>
      </subterm>
      <subterm>
        <variable refvariable="3"/>
      </subterm>
    </mult>
  </def>
</namedoperator>
```

Listing 4.32: Mapping bundled association elements

```
metadata.add(
  TermsPackage.eINSTANCE.getNamedOperator_Parameters(),
  TermsPackage.eINSTANCE.getNamedOperator(),
  TermsPackage.eINSTANCE.getVariableDecl(),
  "parameter",
  null,
  HLPNGFactory.getHLPNGFactory(),
  true);

metadata.add(
  null,
  null,
  TermsPackage.eINSTANCE.getVariableDecl(),
  "variabledecl",
  null,
  HLPNGFactory.getHLPNGFactory());
```

rameter `true` says, that this is a bundled association. The second table entry defines the mappings for variable entries, which is independent of the context, which is why the first to parameters are `null`. We call this a *context independent element mapping*.

This context independent element mapping can be applied in any other context. In combination with another special case of mappings which we call *standard feature*, this is a very powerful mechanism. For example, for `Declarations` and sub-elements for which context independent element mappings exist (in the example, there would be variable declarations, sort declarations, and operator declarations), all these elements should be added to this standard feature. The table entry shown in Listing 4.33 defines the composition `declaration` as the standard feature of `Declarations`. Note

Listing 4.33: Defining a standard feature

```
metadata.add(TermsPackage.eINSTANCE.getDeclarations_Declaration(),
  TermsPackage.eINSTANCE.getDeclarations(),
  TermsPackage.eINSTANCE.getDeclaration(),
  null,
  null,
  null);
```

that there is no mapping to XML here. A standard feature of an element just says that, whenever there comes some context independent element that is not mapped explicitly to a feature, this element should be added to the standard feature of the model. Of course, there should only be one standard feature – otherwise there would be some ambiguities.

4.5.4 Petri net type definitions: Summary and overview

In Sect. 4.5.1–4.5.3, we have seen most of the mechanisms for defining new Petri net types. Basically, a Petri net type definition consists of a new Ecore package where the Petri net type of the PNML core model is extended and the classes for the extended Petri net elements are modelled. Form this model the major parts of the code (model and edit code) can be generated. In the generated code, some manual changes need to be made. The Ecore package needs to follow some modelling principles that are discussed in Sect. 4.5.1.1 and the manual changes are discussed in Sect. 4.5.1.2. All the extensions must be added as *labels* of the respective kind of node of the Petri net.

If a label should not be shown as annotation of the respective element in the graphical editor of the ePNK, this can be achieved by deriving it from the ePNK class `Attribute`. Attributes can be edited in the properties view of the ePNK only. Sect. 4.5.2 discussed an example.

More complex Petri net types might require to also implement a parser and to store the actual information of the label not only as text but also as an abstract syntax tree in the PNML file. Such labels are called structured labels and have been discussed in Sect. 4.5.3.2. In case of complex Petri net types, it might also be necessary to customize the XML representation, of the labels and the concepts of its abstract syntax. To this end, the ePNK allows Petri net types to define a XML mapping, which is discussed in Sect. 4.5.3.4.

For all kinds of nets, it is possible to add additional constraints on top of the Ecore model of the respective type. These constraints are plugged in via the standard mechanisms for EMF Validation. Two examples are discussed in Sect. 4.5.1.4 and Sect. 4.5.3.3. The constrains can either be programmed in Java or can be OCL.

Note that it is possible to extend the class `PetriNet` of the PNML core model (when net labels are needed in the Petri net type). It is also possible to add labels to pages and reference nodes by extending the respective classes in the Ecore model for the new Petri net type.

When an annotation is defined for a page, the question is whether the respective annotation should be shown as a label annotated to the node

page on the super page or whether the label should be shown as a page label on the page itself. The name of a page is shown as an annotation of the page on the super page; by default, an annotation of a page is shown as page labels on the page itself, if there can be multiple annotations of that kind for the page; and it is shown as a label of the page node on the super page, if there can be only one annotation of that kind. But, this can be changed by overriding the method `showLabelOnPage()` of the class `Page` of the respective Petri net type – which requires manual coding again.

4.6 Defining the graphical appearance

For some kinds or Petri nets, some places, transitions or arcs should be shown in a dedicated graphical representation. And the graphical appearance might depend on the context of the respective element – and the graphical appearance might change dependent on the changes of the context of this element. An example are signal arcs, inhibitor arcs, and read arcs in SE-nets, an example of which is shown in Fig. 4.9 again.

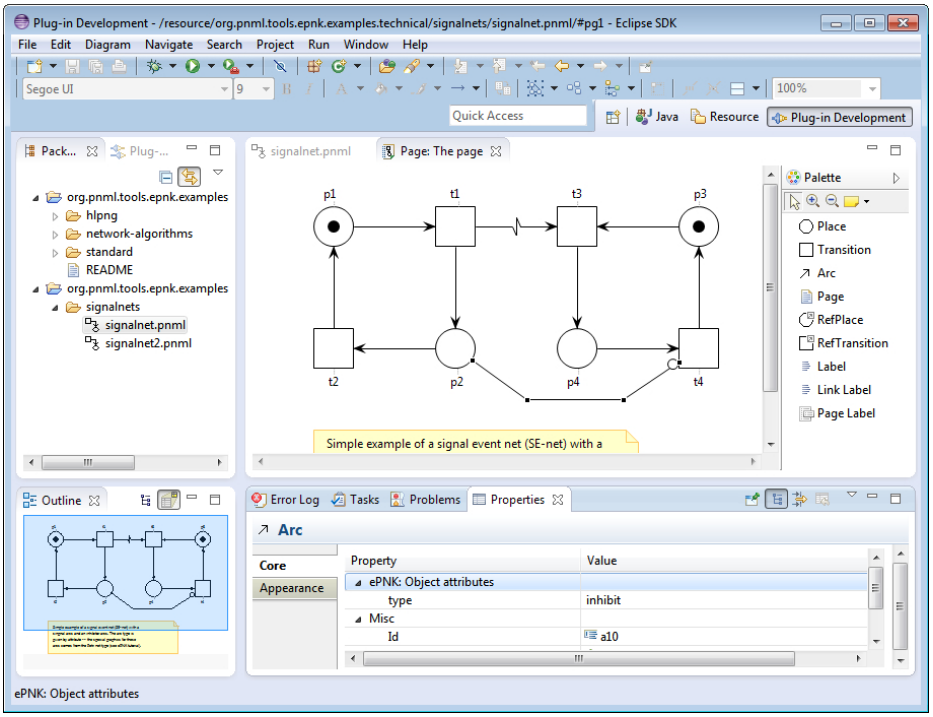

Figure 4.9: A SE-net with its dedicated graphics

In this section, we discuss how such dedicated graphics can be plugged into the ePNK. To this end, we continue the discussion of the projects that implement SE-nets, which was started in Sect. 4.5.2. As you can see from Fig. 4.9, SE-nets have a dedicated graphics for arcs (as signal arc, read arc, or inhibitor arc). But there is also a dedicated graphics for places: the marking is shown by black dots – up to some upper bound – in the respective places.

We start discussing the implementation of the dedicated graphical representation for arcs. To this end, we need to implement a *figure class*, which is the GEF/GMF terminology for the graphically visible elements (view) of a model element in an editor. Listing 4.34 shows the main part of the class `SignalnetArcFigure`, which implements the graphical appearance of the arcs of SE-nets (the class `SignalnetArcFigure` can be found in the package `org.pnml.tools.epnk.pntypes.signalnets.graphics.figures` of plug-in project `org.pnml.tools.epnk.pntypes.signalnets`). This class extends the class `ArcFigure` of the ePNK. In line 3, an enumeration of possible arc types is define, which is private to this class. Note that we do not re-use the enumeration from the model here, but define another enumeration, in order to make the implementation a bit simpler. The current type of the arc is stored as an attribute of this class (line 5). The constructor (lines 7–11) takes the arc (the model element behind this figure) as a parameter; it calls the constructor of the super class `ArcFigure` of the ePNK, which also takes the arc as a parameter, then calculates the current type (by the private method `getType()`, which is shown in List. 4.35) and then properly sets the graphical features by calling the method `setGraphics()`, which is specific to this class.

The method `setGraphic()` changes the graphical features of the arc according to the current type of the arc (lines 21–39). In this example, we change the *decorations* of the arc only; we use the decorations at both ends (source and target). As decorations, we use the usual arrow shaped ones (`ReisigArrowHeadDecoration`[38]), circles (`CircleDecoration`), and flashes (`FlashDecoration`), which are provided by the ePNK. In the method, the variables for the decorations on both ends are initialized to `null` (lines 22-23). Then, dependent on the type of the arc the respective decorations are set. Note that the `FlashDecoration` is attached to the source, but it will actually show up in the middle of the arc (or actually in the middle of the

[38]This name was chosen in honour of Wolfgang Reisig, who insisted on arrow heads in Petri nets being drawn in a very specific way. The implementation of `ReisigArrowHeadDecoration` tries to meat Wolfgang Reisig's standards.

Listing 4.34: The class `SignalnetArcFigure`: main part

```
public class SignalnetArcFigure extends ArcFigure {

  private enum Type { NORMAL, READ, INHIBIT, SIGNAL }

  private Type type;

  public SignalnetArcFigure(Arc arc) {
    super(arc);
    type = getType();
    setGraphics();
  }

  @Override
  public void update() {
    Type oldType = type;
    type = getType();
    if (oldType != type) {
      setGraphics();
  } }

  private void setGraphics() {
    RotatableDecoration targetDecorator = null;
    RotatableDecoration sourceDecorator = null;

    if (type == Type.READ) {
      targetDecorator = new ReisigsArrowHeadDecoration();
      sourceDecorator = new ReisigsArrowHeadDecoration();
    } else if (type == Type.INHIBIT) {
      targetDecorator =  new CircleDecoration();
    } else if (type == Type.SIGNAL) {
      sourceDecorator = new FlashDecoration();
      targetDecorator = new ReisigsArrowHeadDecoration();
    } else {
      targetDecorator = new ReisigsArrowHeadDecoration();
    }

    this.setTargetDecoration(targetDecorator);
    this.setSourceDecoration(sourceDecorator);
  }

  ...
```

first segment of the arc) by the specific way it is is implemented. The reason for this choice is that there can be at most one decoration at each end of a *connection*. Since signal arcs have two decorations, the flash and the arrow head, one needs to be at the source end of the connection. In the end, the only thing that is necessary to do is actually setting the decorations – note that calling the respective methods with `null`, means that there is no decoration for that connection.

Note that changes in the underlying model might make changes in the graphical appearance necessary. Such a change could be an explicit change of the type of the arc by the end user or just reconnecting an arc to a different kind of element. Whenever such a change happens, the ePNK notifies the figure of the affected model element by calling the method `update()`, which is specific to all extensible figure classes of the ePNK. It should be overridden by the extending classes. Lines 14–19 of List. 4.35 show the implementation of this method in our example. The type of the arc is computed again; if it changed, the `setGraphics()` method is called again.

This is all there is to do for implementing another appearance of an arc. Of course, this figure still needs to be plugged in, which is discussed later. In the update method, you could do all kinds of other changes such as changing the colour of the arc (`setForeGroundColor()` or the line style (`setLineStyle()`) – and many things more.

If you need other decorations than the ones that come with the ePNK, you can implement them yourself. But, we do not discuss this here since this is a GMF or, actually, an Eclipse draw2d concept. A look at the implementation of the ePNK decorations might give you a clue.

Listing 4.34 shows the last part of the class `SignalnetArcFigure`: the implementation of method `getType()`. This is mostly straightforward: computing the type based on the information of the arc underlying this figure. The only surprise might be the initial type check of `this.arc` for `Arc`. The reason is that `this.arc` refers to a final attribute of `ArcFigure` of the ePNK, which refers to the `Arc` of the PNML core model, whereas in the class `SignalnetArcFigure`, we need to refer to the `Arc` of the SE-net package – which has the same name, but in a different package.

In the above example, we have changed the appearance of the arc on a very high level of programming, by changing the attributes of the figure. And if the desired graphical appearance can be achieved this way, this is the recommended way of doing this. In some cases, however, changing the attributes of the figure it not enough – we rather would need to "draw" some additional things. This can be done by using a different strategy for extending the figure: overriding the `fillShape()` or `outlineShape()`

Listing 4.35: The class `SignalnetArcFigure`: compute type

```
  ...

  private Type getType() {
    if (this.arc instanceof Arc) {
      ArcType arctype = ((Arc) arc).getType();
      if (arctype != null) {
        switch (arctype.getText().getValue()) {
        case ArcTypes.READ_VALUE:
          return Type.READ;
        case ArcTypes.INHIBIT_VALUE:
          return Type.INHIBIT;
        }
      } else {
        Node source = arc.getSource();
        Node target = arc.getTarget();
        if (source instanceof TransitionNode &&
          target instanceof TransitionNode) {
          return Type.SIGNAL;
        }
      }
    }
    return Type.NORMAL;
  }

}
```

methods. We explain this strategy by another example: showing the initial marking by a respective number of black tokens in the place. Listing 4.36 shows the class `SignalnetPlaceFigure`, which implements this graphical appearance. The `update()` method just informs the figure that it should repaint itself[39] when something has changed. The actual appearance is now defined by overriding the method `fillShape()`. In this method, first, all the normal drawing of the place is done by calling the same method of the `super` class. After that, the marking of the place is computed[40]. If the marking is between 1 and 4, the respective number of tokens are drawn in the client area of the place. To this end, the drawing methods on the `graphics` object are used. Depending on the number of tokens, the appropriate positions are chosen (note that for space reasons, we omit the code for drawing four tokens). If there are more than four tokens, they are not represented as black circles anymore. They are "drawn" as a string representing the number of tokens.

Note that you could do more things and could also use some other low-level methods of figures to do that. But, we do not discuss that here. You might get some more inspiration my looking at another tutorial which implements some more exotic appearances of arcs, places and transitions in project `org.pnml.tools.epnk.extensions.tutorial.types`; the resp. figures can be found in the in package `org.pnml.tools.epnk.extensions.tutorial.types.arctypes.graphicalextensions.figures`.

At last we need to make the new figures defined for SE-nets known to the ePNK – we need to plug them in. This is done by implementing and plugging in a factory for these figures. The factory for the graphical extension for SE-nets is shown in List. 4.37. It extends the abstract ePNK class `GraphicalExtension`. The first method (line 4–8) defines, for which types of Petri nets this extension provides some graphics – as a list of classes of the respective Petri net types. In our example, this is the class representing SE-nets – obtained from the automatically generated package class. The second method (line 11–18) defines for which kinds of elements there is a specific graphics – again represented as a list of class objects. In our example, these are the classes representing the arc and the place of SE-nets. At last, there are two methods, that create the respective figure object for an element by using the respective constructors of the figure classes. Note that `GraphicalExtension` has also methods for creating a figure for transitions

[39] We could have done that in a slightly smarter way so that repaint is called only if the marking has changed – as for the arcs.

[40] Since this requires some navigation in the model, this is delegated to a separate method `getMarking()`, which is not discussed here.

Listing 4.36: The class `SignalnetPlaceFigure`

```
public class SignalnetPlaceFigure extends PlaceFigure {

  public SignalnetPlaceFigure(Place place) {
    super(place);
  }

  public void update() {
    this.repaint();
  }

  protected void fillShape(Graphics graphics) {
    super.fillShape(graphics);
    Rectangle rectangle = this.getClientArea();

    int m = 0;
    if (place instanceof Place)
      m = getMarking((Place) place);
    int cx = rectangle.x + rectangle.width/2;
    int cy = rectangle.y + rectangle.height/2;
    if (m == 0) {
      return;
    } else if (m == 1) {
      graphics.setBackgroundColor(getForegroundColor());
      graphics.fillOval(cx-6, cy-6, 12, 12);
    } else if (m == 2) {
      graphics.setBackgroundColor(getForegroundColor());
      graphics.fillOval(cx-11, cy-11, 12, 12);
      graphics.fillOval(cx, cy, 12, 12);
    } else if (m == 3) {
      graphics.setBackgroundColor(getForegroundColor());
      graphics.fillOval(cx-6, cy-13, 12, 12);
      graphics.fillOval(cx-13, cy, 12, 12);
      graphics.fillOval(cx+1, cy, 12, 12);
    } else if (m == 4) {
      ...
    } else {
      graphics.drawString(""+m, cx-5, cy-7);
  } }

  private int getMarking(Place place) { ... }
}
```

Listing 4.37: The factory class `SignalnetGraphics`

```
public class SignalnetGraphics extends GraphicalExtension {

  @Override
  public List<EClass> getExtendedNetTypes() {
    ArrayList<EClass> list = new ArrayList<EClass>();
    list.add(SignalnetsPackage.eINSTANCE.getSignalNet());
    return list;
  }

  @Override
  public List<EClass> getExtendedNetObjects(EClass netType) {
    ArrayList<EClass> list = new ArrayList<EClass>();
    if (netType.equals(SignalnetsPackage.eINSTANCE.getSignalNet())) {
      list.add(SignalnetsPackage.eINSTANCE.getArc());
      list.add(SignalnetsPackage.eINSTANCE.getPlace());
    }
    return list;
  }

  @Override
  public ArcFigure createArcFigure(Arc arc) {
    if (arc instanceof org.pnml.tools.epnk.pntypes.signalnets.Arc) {
      return new SignalnetArcFigure(
          (org.pnml.tools.epnk.pntypes.signalnets.Arc) arc);
    }
    return null;
  }

  @Override
  public IUpdateableFigure createPlaceFigure(Place place) {
    if (place instanceof
        org.pnml.tools.epnk.pntypes.signalnets.Place) {
      return new SignalnetPlaceFigure(
          (org.pnml.tools.epnk.pntypes.signalnets.Place) place);
    }
    return null;
  }

}
```

and other kinds of nodes – but we do not need to override them here since our extension does not provide special graphics for them.

Actually the class `GraphicalExtension` has some more methods, which define priorities for the graphical extensions – in case more than one graphical extension is plugged in and applies to the same element. And it can also be defined, whether a graphical extension should apply to all subtypes of a Petri net type or not. The meaning of the methods is documented as Java doc comments for the methods in the interface for the factory `IGraphicalExtension`.

At last, the graphical extension needs to be plugged in to the ePNK. The relevant part of the "plugin.xml" of the project is shown in List. 4.38. This is straightforward. The attribute "point" of the extension refers to the ePNK extension point `org.pnml.tools.epnk.diagram.graphics`, and the class attribute of the graphicsextension refers to the factory of List. 4.37.

Listing 4.38: Plugging in the graphical extension

```
<extension
    id="org.pnml.tools.epnk.pntypes.signalnets.graphics"
    name="Signal event net graphical extensions"
    point="org.pnml.tools.epnk.diagram.graphics">
  <graphicsextension
    class="org.pnml. ... .signalnets.graphics.SignalnetGraphics"
    description="Special graphics for ... signal event nets">
  </graphicsextension>
</extension>
```

With this extension installed, the SE-nets should now look like the one in Fig. 4.9.

4.7 Adding tool specific information

As discussed in Sect. 2.2.1, the PNML allows tool specific information to be added to all elements of Petri nets – indicated by the special XML element `<toolspecific>`. The ePNK reads and writes any tool specific information, and, in principle, the contents of these tool specific extensions could be accessed and modified via the class `AnyType`, which is defined in the plug-in `org.eclipse.emf.ecore`. But this is tedious and, basically, means navigating in the element's XML structure.

Therefore, the ePNK provides an extension point for plugging in tool specific extensions, so that they can be accessed and modified via an API specific to the extension which can be defined in terms of a model. We will discuss how to use this extension point by the help of an example: the token positions, which is a tool specific extension mandated by ISO/IEC 15909-2:2011. We have seen an example already in Fig. 2.3 and Listing 2.1 on page 10.

This tool specific extension is defined in the project `org.pnml.tools.epnk.toolspecific.tokenpositions`. Most of this code in this project as well as the "plugin.xml" was automatically generated by EMF from the Ecore model "Tokenpositions.ecore" in the folder "model". This model is shown in Fig. 4.10. The new classes are `PNMLToolInfo` and `Tokengraphics`.

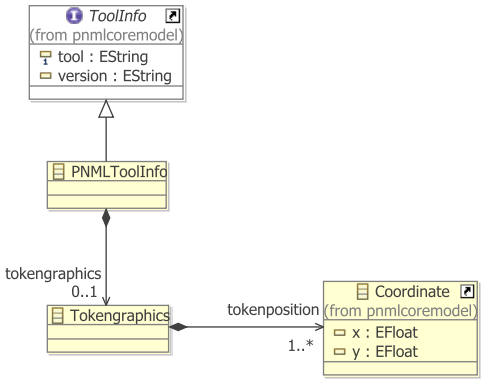

Figure 4.10: The model for tool specific extension tokenpositions

The class `PNMLToolInfo` represents the actual tool specific information: it must implement the PNML core model interface `ToolInfo`. The actual contents of this tool specific information is `Tokengraphics`, which consists of one or many coordinates; the class `Coordinate` is re-used from the PNML core model.

From this model, the code can be generated in the same way as described in Sect. 4.5.1.2. First, the "genmodel" must be created, and from the "genmodel", the model code and the edit code must be generated.

After the code generation, the only thing left to do is to manually create a factory for this tool specific extension, and use this factory for plugging it into the ePNK. The factory for our extension is shown in Listing 4.39. The factory implements the ePNK interface `ToolspecificExtensionFactory`, which consists of four methods. The two methods `createToolInfo()` cre-

Listing 4.39: Factory for the tool specific extension

```
package org.pnml.tools.epnk.toolspecific.tokenpositions.factory;

import org.pnml.tools.epnk.pnmlcoremodel.ToolInfo;
import org.pnml.tools.epnk.toolspecific.extension.
   ToolspecificExtensionFactory;
import org.pnml.tools.epnk.toolspecific.tokenpositions.
   TokenpositionsFactory;


public class TokenpositionsExtensionFactory
    implements ToolspecificExtensionFactory {

  private final static String toolname =     "org.pnml.tool";
  private final static String toolversion =  "1.0";

  public ToolInfo createToolInfo(String tool, String version) {
    // ToolInfo object does not depend on these values:
    return createToolInfo();
  }

  public ToolInfo createToolInfo() {
    return TokenpositionsFactory.eINSTANCE.createPNMLToolInfo();
  }

  public String getToolName() {
    return toolname;
  }

  public String getToolVersion() {
    return toolversion;
  }

}
```

ate an instance of this tool specific extension; the method with the two String parameters, `tool` and `version` is used, when the tool name and version are given, which might return instances of different classes – in our example, however, the version number is irrelevant. The two other methods, must return the tool name for that extension and its version, which, in our example, are encoded as constants.

Listing 4.40 shows the fragment of the "plugin.xml" that is needed to plug in the token position extensions to the ePNK. In addition to the name and the id, there is an attribute `class` that defines the factory for the tool specific extension; this class must implement the interface `Toolspecific ExtensionFactory`. Moreover, there is a brief description of this extension.

Listing 4.40: Plugging in the token position extension

```
<extension
  id="org.pnml.tools.epnk.toolspecific.tokenpositions"
  name="Token Positions"
  point="org.pnml.tools.epnk.toolspecific">
  <type
    class="org.pnml. ... .factory.TokenpositionsExtensionFactory"
    description="The tool specific extension for token positions">
  </type>
</extension>
```

Note that the ePNK does not provide any way yet of explicitly defining the XML syntax of these extensions. The standard XMI serialisation will be used – which is compliant with ISO/IEC 15909-2:2011 for tool specific extensions. Eventually, the ePNK might provide a mapping mechanism similar to the one for Petri net types.

4.8 Overview of the ePNK and its projects

In this section, we give a brief overview of the different parts of the ePNK, the project structure and where to look for different kinds of functionality in the ePNK API an its projects. As mentioned earlier, developers should not change anything in these projects. Anyway, this overview should help to better understand the ideas behind the ePNK, the necessary dependencies (that need to be included in new projects via the "plugin.xml") and the functions that are available in the ePNK API, which could be used by developers in their extensions. Note that we do not discuss the details of

the API here. In particular, we do not discuss the API concerning the code that is generated from the Ecore models (model and edit code) since it is mostly straightforward. Section 4.3.4.1 gives a brief overview of the main principles behind the model code that is generated from Ecore models; for more information, we refer to the EMF book [2].

Like all extensions of Eclipse, the ePNK is organized in many Eclipse projects, which together make the ePNK. Most of these projects are so-called plug-in projects; and these are the ones most relevant for developers, since these are the projects to look up the API and to which extensions need to refer (in form of dependencies). In addition, there are some projects, which contain documentation only (like this manual), there are some projects that do not contain any code, but from which other projects are generated; and there are so-called features, which define collections of plug-in projects in order to deploy them. And there is a project for generating the ePNK update site from the features.

In this manual, we focus on the plug-in projects of the ePNK, the most important of which are listed below. We start with an overview of the ePNK core projects, which make up the framework of the ePNK[41]:

`org.pnml.tools.epnk`:
: This is the core project of the ePNK. In this project, you will find the PNML core model, some additional models, and the *model code*, that was generated from them. All the models, can be found in the folder "model". The PNML core model is contained in `PNMLCoreModel.ecore`; in order to avoid clutter in the graphical diagram, the core model is actually split up into three separate diagrams: `PNMLCoreModel.ecorediag` contains the most important concepts; `PNMLCoreModelGraphics.ecorediag` contains the graphical features of PNML; and `PNMLCoreModelProxies.ecorediag` contains some extensions to the PNML core model that are necessary to maintain labels in the graphical editor of the ePNK by so-called *label proxies* and *page label proxies*. These proxy elements, however, are not of any concern for normal developers.

  There are four other models in this project: `PNMLDataTypes.ecore` defines the *data types* for non-negative and positive numbers, which are used instead of the respective XML Schema Data Types of ISO/IEC 15909-2. `PNMLStructuredPNTypeModel.ecore` defines the concepts of structured Petri net types (see Sect. 4.5.3.2). `Serialisation.ecore`

[41] Technically, all these plug-in projects are part of the ePNK features `org.pnml.tools.epnk.core` and `org.pnml.tools.epnk.extensions.basic`.

provides some general structure that is used for the XML serialisation of so-called association elements (see Sect. 4.5.3.4). `PNMLPageDiagramInfo.ecore` is the model for storing the GMF diagram information for pages as ePNK tool specific information in PNML models, which are not of concern for normal developers.

This project provides also some *convenience classes* in package `org.pmm.tools.epnk.helpers`, which might be helpful in practice. The class `FlatAccess` allows handling a Petri net that is distributed over several pages as if it was flat (see Sect. 4.3.3 for an example). Another convenience class is `NetFunctions`, which provides many static methods for finding out to which net an element belongs, what the type of this net is, and for obtaining lists of elements of some kind of a given Petri net.

In this plug-in, also the two extension points of the ePNK are defined: one for defining new Petri net types (PNTD), another for defining new tool specific extensions.

In addition to the model code, also the so-called edit code and editor code are generated from these models, which together define the tree editor for PNML (see below).

Note that, in this project, also the constraint context and the constraint category `org.pnml.tools.epnk.validation`, to which all other constraints for new Petri net types should be added, are defined here (see Sect. 4.5.1.4 and 4.5.3.3).

`org.pnml.tools.epnk.edit:`
: This project contains the edit code that was generated from the models in project `org.pnml.tools.epnk`. Though most of this code was automatically generated from the models, there are several manual changes, that enable generically dealing with plugged in Petri net type definitions.

  Moreover, the generated standard EMF images in the folder `icons` were replaced by nicer ones.

`org.pnml.tools.epnk.editor:`
: This project contains the editor code for the EMF tree editor for PNML that was generated from the models in project `org.pnml.tools.epnk`. In this project, there are only a few, but crucial extensions, that made it possible to integrate the EMF tree editor with the

graphical editor for pages (see the plug-in project `org.pnml.tools.epnk.diagram` below).

`org.pnml.tools.epnk.pntypes`:
: This project contains the model and the generated model code for P/T-nets (`PTNet`), as well as the extension that plugs in this type to the ePNK. The model code is completely generated from the model `PTNet.ecore` in the folder "model", except for two changes in class `PTNetImpl` as discussed in Sect. 4.5.1.

`org.pnml.tools.epnk.pntypes.edit`:
: This is the project with the edit code that was generated from the Ecore model `PTnet.ecore` of project `org.pnml.tools.epnk.pntypes`. There are no manual changes in the generated code – only the icons in the folder `icons` were replaced by nicer ones.

`org.pnml.tools.epnk.toolspecific.tokenpositions`:
: In this project, the tool specific extension for token positions (as defined in ISO/IEC 15909-2) is defined (see Sect. 4.7). The model code and the edit project `org.pnml.tools.epnk.toolspecific.tokenpositions.edit` was generated (which does not contain any manual changes – not even nicer icons) from the model `Tokenposition.ecore`.

  Note that the ePNK does not take the information of these token positions into account in the graphical representation of places. The only reason they are defined in the ePNK is that ISO/IEC 15909-2:2011 mandates them – and we use this extension as an example to show how to define tool specific extensions in Sect. 4.7.

`org.pnml.tools.epnk.actions`:
: This project defines the standard actions of the ePNK, which are the pop-up menus for adding missing ids and for linking the labels of structured Petri net types (see Sect. 4.5.3.2).

  Moreover, the classes `AbstractEPNKAction` and `AbstractEPNKJob` are defined in this project, which are convenience classes to make it easier to define functions for the ePNK that run in the background (see Sect. 4.3.3).

`org.pnml.tools.epnk.diagram`:
: This project contains the code for the GMF-generated graphical editor for pages of Petri nets. This code was generated from the GMF models in project `org.pnml.tools.epnk.gmf`. But, there are major manual

changes for making this graphical editor generic and for integrating it with the tree editor for PNML (see project `org.pnml.tools.epnk.editor`).

The package `org.pnml.tools.epnk.gmf.extensions.graphics` and its sub-packages `decorations` and `figures` are relevant for developers, who want to contribute specific graphical appearances for some Petri nets. Here you find the factory and the figures of the ePNK, which need to be extended for customizing the graphical appearance; and you can use the predefined ePNK decorations for that purpose (see Sect. 4.6 for more details).

`org.pnml.tools.epnk.gmf.integration:`
This project defines the pop-up menus for starting the graphical editor on a page that is selected in a tree editor or in a graphical editor.

`org.pnml.tools.epnk.annotations:`
This project defines the infrastructure for annotating Petri nets by applications. Up to now, the annotations provide the most basic concepts only – they will be extended in the future.

A simple example of annotating the context of transitions was discussed in Sect. 4.4.

`org.pnml.tools.epnk.applications:`
This project provides the interfaces and classes for implementing and starting applications on Petri nets, which was briefly discussed in Sect. 4.4.

`org.pnml.tools.epnk.applications.view:`
This project implements the applications view, that shows all currently running implementations. Developers would typically not directly use this project.

Next, we discuss some of the net types, and functions and applications, which we had discussed in this manual as a tutorial[42]:

`org.pnml.tools.epnk.functions.tutorials:`
This project contains the functions that were discussed in Sect. 4.3.1 and Sect. 4.3.2, which can serve as a guideline for defining own extension projects.

[42]Technically, these plug-in projects come from the ePNK feature `org.pnml.tools.epnk.extensions.tutorial`.

`org.pnml.tools.epnk.functions.modelchecking`:
: This project contains the model checker extension for the ePNK that is discussed in Sect. 4.3.3. Note that the project `MCiE` is used in this model checker extension only (MCiE is not relevant for anything else in the ePNK – except if you want to implement your own model checker based on MCiE).

`org.pnml.tools.epnk.tutorials.applications`:
: This project implements an example of an application using annotations. It contains the code for the transition context application which was discussed in Sect. 4.4.

`org.pnml.tools.epnk.pntypes.signalnets`:
: This project is the plug-in project implementing the Petri net type definition for SE-nets (see Sect. 4.5.2 and the graphical extensions for SE-nets (see Sect. 4.6).

`org.pnml.tools.epnk.pntypes.signalnets.edit`:
: This plug-in project contains the edit code, which is generated from the model of SE-nets – without any manual changes

The plug-in projects with the prefix `org.pnml.tools.epnk.pntypes.hlpng` resp. `org.pnml.tools.epnk.pntypes.hlpngs` together are used for defining high-level Petri nets (HLPNGs). The following list gives an overview in a bottom up way – ending with the actual Petri net type definition for HLPNGs:

`org.pnml.tools.epnk.pntypes.hlpngs.datatypes`:
: In this project, all the models that define the concepts of sorts, operators, variables and terms for HLPNGs are contained. In particular, there is a model `HLPNGDataTypes.ecore` for the general structure of terms, declarations and the built-in sorts and operators that occur in all versions of HLPNGs. And there are many more models and diagrams for specific versions of HLPNGs (Clauses 5.3.2–5.3.12 of ISO/IEC 15909-2:2011).

`org.pnml.tools.epnk.pntypes.hlpngs.datatypes.concretesyntax`:
: This project is an Xtext project that defines the grammar for the concrete syntax of the different labels of HLPNGs, from which a parser is generated. Here, we do not discuss the details of generating the parser. The manually written class `HLPNGParser` accesses the automatically

generated parsing operations and provides methods for parsing every kind of label of HLPNGs. The other important manually written class is `HLPNGLinker`, which provides the global Linker for labels (as discussed in Sect. 4.5.3.2).

Note that the plug-in project ending with `concretesyntax.ui` was automatically created when the Xtext project was created by a wizard; it provides a stand-alone textual editor for the labels of HLPNGs. Since this stand-alone editor is not needed for the ePNK, this project is not part of the standard deployment of the ePNK.

`org.pnml.tools.epnk.pntypes.hlpng.pntd:`
This project actually combines all the parts discussed above into a Petri net type definition for HLPNGs. The main model is `HLPNG Definition.ecore`, which was shown in Fig. 4.6. The other model `HLPNGSerialisation.ecore` defines the auxiliary classes that temporarily store XML elements that represent associations and are used and referred to in the XML mappings (see Sect. 4.5.3.4) .

The global linker `HLPNGLinker` from the project `org.pnml.tools.epnk.pntypes.hlpngs.datatypes.concretesyntax` above is made available in the Petri net type by manually implementing the method `getLinker()` in class `HLPNGImpl`. The `parse()` method for the different structured labels are also manually implemented – they refer to the different `parserXXX()` methods of class `HLPNGParser` from the project `org.pnml.tools.epnk.pntypes.hlpngs.datatypes.concretesyntax` above.

`org.pnml.tools.epnk.helpers.unparse:`
This project implements a serialisation function that transforms the abstract syntax of HLPNG labels into the concrete syntax used for HLPNGs in the ePNK. This is mostly relevant for the end users – who want to generate the concrete syntax for the labels of HLPNGs (see Sect. 3.5.2 on page 40).

But, in some cases this label serialiser might also be relevant for developers that want to show terms of HLPNGs to the end user (the simulator for high-level nets is an example).

`org.pnml.tools.epnk.applications.hlpng.simulator:`
This is the plug-in project implementing the basic simulator for HLPNGs. We do not discuss the implementation in this manual, we discussed the simulator only from the end users' point of view in

Sect. 3.6.3. You will find many more details in the master's thesis that implemented it [21].

4.9 Deploying extensions

In this section, we will briefly discuss how own extensions of the ePNK could be deployed, so that others can use it. Typically, an extension comprises several plug-in projects. In order to combine them, Eclipse provides a special kind of project, which is called a *feature* – and this is the unit in which Eclipse extensions should be deployed.

A feature in turn, can be used in an Eclipse *update site project* which can be used to create your own *update site*, so that your plug-ins (resp. features) could be installed from this site, similar to the way you installed the ePNK.

Since features and update sites are standard Eclipse concepts, we do not explain the details here. For now, looking up the keywords "feature" and "update site" in the Eclipse help (or googling for them) should be enough.

If you have a feature for the ePNK that might be interesting for a wider audience, you can also contact us, so that we can make it available via the ePNK update site.

Chapter 5

Experience and outlook

With version 1.0.0, the ePNK has reached a mature state and it should be useful for end users who want to use its graphical editor for creating PNML files and for using the simple function of the ePNK as they are. The ePNK should be even more useful for developers who want to implement new Petri net types and new functionality. In particular, it should be an ideal platform for scientists who quickly want to test new functions and still want a graphical editor that is nicely embedded to an IDE. Actually, we use the ePNK ourselves for implementing the tool support for the Event Coordination Notation (ECNO) [16].

Some of the plans for future extensions of the ePNK are discussed in Sect. 5.2. Before discussing the future plans, we briefly discuss the past in Sect. 5.1: the experiences with developing the ePNK in a model based way and in particular with using EMF, GMF, and some related technologies

5.1 Experiences with MBSE

There are many Petri net tools out there already. Therefore, implementing yet another one needed some additional motivation. When developing the ePNK, this additional motivation was to gain some more experience with the use of EMF and model-based development. To this end, we kept a detailed log of how much time was spent on which parts of the development (up to the first major release of version 0.9.1, the log accounts for minutes).

Eventually, we might break down the time and experiences made in more detail. Here we give an overview of the major steps and the rough time spent on the major parts of the ePNK only:

40h The (roughly) first 40 hours were spent on making a first model of

the PNML core model and implementing the extensions necessary for plugging in new Petri net types and for hooking into the serialisation mechanisms of Eclipse and EMF, so that it could be configured – and on using this new configuration mechanism for implementing the PNML syntax for serialisation instead of standard XMI. Of these 40 hours, about 10 hours were spent on debugging Eclipse's and EMF's serialisation mechanism in order to understand this serialisation mechanism, which unfortunately is not very well documented.

After these first 40 hours, a basic version of the ePNK was working – not supporting HLPNGs yet and without any graphical editor.

20h The next 20 hours were spent on implementing the framework for tool specific extensions – and ignoring unknown tool specifc extensions, as well as on implementing the validation mechanisms and most of the constraints for the PNML core model[1].

50h About 50 hours were spent on implementing HLPNGs, which includes making the models for most of the sorts and operators of HLPNGs, implementing a type system for checking correct typing, and for validating correctness, and implementing parsing and linking functions. These 50 hours include the time spent on adjusting the mechanisms of the ePNK so that it could deal with parsing and linking structured Petri net types. The parser itself is based on Xtext.

Implementing a basic version of HLPNGs with only a few but representative sorts and operators took roughly 20 hours.

80h Implementing the GMF editor in a generic way and properly integrating it with the EMF tree editor, and the parsing mechanisms for structured labels took about 80 hours. This time includes a lot of time investigating and experimenting with different options in achieving this.

30h The remaining 30 hours were spent on implementing some functions as examples (used for the tutorials), on adding some of the last remaining sorts to the definition of HLPNGs, as well as on cleaning up the code and fixing some errors.

[1]This concerns not only the constraints that are explicitly formulated as OCL constraints in the models of ISO/IEC 15909-2; this concerns all the constraints that are stated somewhere in the text of the standard, such as forbidden cycles between reference nodes, etc.

Altogether, it took about 5 $\frac{1}{2}$ weeks working time (spent scattered over about 7 month) to implement version 0.9.1 of the ePNK, which was the first stable version of the ePNK. The core part of the ePNK was implemented in 60 hours. About the same time went into implementing HLPNGs. The implementation of the graphical editor took a major part of the time (80 hours). The reason for that was that GMF itself was not made for building generic editors, and we had to find ways of bringing genericity into GMF – and we had to work around several GMF problems and quirks. Still, using GMF might have saved us a significant amount of time considering the overall functionality that we get for free by a GMF generated graphical editor – and its smooth integration with the Eclipse IDE.

The overall experience was that the parts of the ePNK that concern EMF only worked very smoothly and the use of EMF significantly sped up the development process – for a developer who has some experience with EMF and some of its more advanced concepts already. Working with GMF was more tedious and required much more experience and much more endurance – using the debugger digging in the inner workings of GMF in order to find out how some things work. This is partly due to the fact that GMF was lacking the concepts for generic editors; but, genericity aside, GMF requires much more experience than EMF in order to use it with benefit.

5.2 Future plans

The ePNK can be and is used for different kinds of applications – and it makes it easy to quickly implement new Petri net types and new functions and application. For making these functions more usable and also for easing the fast creation of Petri nets with the ePNK editors, some extension of the ePNK would be useful.

In this section, we give an overview of some extensions that are planned for the ePNK in the future. These extensions concern ePNK's flexibility and ease of use as well as some additional extension mechanisms. The order indicates some priorities, and might be roughly the order in which the features are implemented – but, since nobody is paid for the work, it needs to be seen how things turn out:

- Right now, the ePNK does not "know" the functions and applications that are available for the ePNK. These are plugged in to the Eclipse platform as actions or commands – and initiated by the Eclipse platform. It would be nice if there was an ePNK view that would, for a

selected Petri net, show all the available functions and applications, which could be started from there by a single mouse click.

To this end, an explicit extension point to plug in functions and applications for the ePNK needs to be defined. This extension point could also provide some means to give some meta information about the function or application, saying to which net types it applies and on its characteristics.

Such a plug-in mechanism along with a view for showing the functions and applications available for the selected Petri net will be implemented in a future version of the ePNK.

- Right now, the annotations are shown by marking the elements with a red overlay. Changing this is possible but quite complicated.

  A future version of the ePNK will support a mechanism for applications to provide a presentation descriptor, which defines how annotations should be shown (with some reasonable default implementation). And this descriptor should also be able to define how the end user can interact with the annotated elements and define actions that are initiated by that.

- Right now, adding all the needed labels to a Petri net element of a more complex net types such as HLPNGs with the ePNK editor is quite tedious: First, each label must be created, then the label must be connected to the element, and its type must be selected.

  In a future version of the ePNK, an action will be installed that, e. g. on a double click on a node, will add all the labels required by the Petri net type for that node.

- Right now only the ePNK tree editor will show the correct "dirty flag" after a change of the model. And the PNML document can be saved only from this tree editor.

  In a future version of the ePNK, the "dirty flag" will be updated in all editors that are open on that document and saving (in particular with CTRL-S) will work from any of these editors.

- Right now, the mapping of the model elements of the ePNK and of Petri net types to their XML representation must be programmed – if it needs to be changed. This means programming large tables, which is boring, tedious and error prone.

In a future version of the ePNK, we might define an extension point for such XML mapping that directly takes a table instead of "programming the table".

- Right now, the ePNK comes with its own specific and fixedly implemented concrete syntax for the labels of HLPNGs. Since this syntax is not mandated by ISO/IEC 15909-2, different tools might use different concrete syntax for these labels.

  It would be nice, if different versions of such concrete syntax for labels of HLPNGs could be plugged in to the ePNK, from which the user could choose.

- Up to now, the ePNK supports basic and structured PNML [29] only. Modular PNML is not supported by the ePNK yet – nor is it part of ISO/IEC 15909-2:2011.

  In the long term, the ePNK should support also modular PNML (which was proposed in [29] and some aspects worked out in more detail in [14] already). Implementing the EMF models would actually not be a big issue – implementing the graphical editor of the ePNK so that it works for both PNML as of ISO/IEC 15909-2:2011 as well as for modular PNML is the actual challenge here – and the reason for not having started on this endeavor yet.

Bibliography

[1] J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The Petri Net Markup Language: Concepts, technology, and tools. In W. van der Aalst and E. Best, editors, *Application and Theory of Petri Nets 2003, 24th International Conference*, volume 2679 of *LNCS*, pages 483–505. Springer, June 2003.

[2] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, 2nd edition edition, Apr. 2006.

[3] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.

[4] E. Clayberg and D. Rubel. *Eclipse Plug-ins*. The Eclipse Series. Addison-Wesley, 3rd edition edition, 2008.

[5] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. The Eclipse Series. Addison-Wesley Professional, Mar. 2009.

[6] L. Hillah, E. Kindler, F. Kordon, L. Petrucci, and N. Treves. A primer on the Petri Net Markup Language and ISO/IEC 15909-2. In K. Jensen, editor, 10th *Workshop on Coloured Petri Nets (CPN 09)*, pages 101–120, Oct. 2009.

[7] L.-M. Hillah, F. Kordon, L. Petrucci, and N. Trèves. PNML Framework: An extendable reference implementation of the Petri Net Markup Language. In J. Lilius and W. Penczek, editors, *Petri Nets*, volume 6128 of *Lecture Notes in Computer Science*, pages 318–327. Springer, 2010.

[8] ISO/IEC. Systems and software engineering – High-level Petri nets – Part 2: Transfer format, International Standard ISO/IEC 15909-2:2011, Feb. 2011.

[9] K. Jensen and L. M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer-Verlag, 2009.

[10] M. Jüngel, E. Kindler, and M. Weber. The Petri Net Markup Language. *Petri Net Newsletter*, 59:24–29, Oct. 2000.

[11] M. Jüngel, E. Kindler, and M. Weber. Towards a generic interchange format for Petri nets – position paper. In R. Bastide, J. Billington, E. Kindler, F. Kordon, and K. H. Mortensen, editors, *Meeting on XML/SGML based Interchange Formats for Petri Nets*, pages 1–5, June 2000.

[12] E. Kindler. Der Petrinetz-Kern: Ein Traum wird wahr. In H. Ehrig, W. Reisig, and H. Weber, editors, *Move-On-Workshop der DFG-Forschergruppe Petrinetz-Technologie*, pages 121–124. Technische Universiät Berlin, 1997.

[13] E. Kindler. The Petri Net Markup Language and ISO/IEC 15909-2: Concepts, status, and future directions. In E. Schnieder, editor, *Entwurf komplexer Automatisierungssysteme, 9. Fachtagung*, pages 35–55, May 2006. invited paper.

[14] E. Kindler. Modular PNML revisited: Some ideas for strict typing. In *Proc. AWPN 2007, Koblenz, Germany*, Sept. 2007.

[15] E. Kindler. Model-based software engineering and process-aware information systems. In K. Jensen and W. van der Aalst, editors, *Transactions on Petri Nets and Other Models of Concurrency II: Special Issue on Concurrency in Process-Aware Information Systems*, volume 5460 of *LNCS*, pages 27–45. Springer-Verlag, 2009.

[16] E. Kindler. Modelling local and global behaviour: Petri nets and event coordination. *Transactions on Petri Nets and Other Models of Concurrency*, 6:71–93, 2012.

[17] E. Kindler and J. Desel. Der Traum von einem universellen Petrinetz-Werkzeug — Der Petrinetz-Kern. In J. Desel, A. Oberweis, and E. Kindler, editors, *3. Workshop Algorithmen und Werkzeuge für Petrinetze*, number 341 in Forschungsberichte. Institut AIFB, Universität Karlsruhe, Oct. 1996.

[18] E. Kindler and W. Reisig. Algebraic system nets for modelling distributed algorithms. *Petri Net Newsletter*, 51:16–31, Dec. 1996.

[19] E. Kindler, W. Reisig, H. Völzer, and R. Walter. Petri net based verification of distributed algorithms: An example. *Formal Aspects of Computing*, 9:409–424, 1997.

[20] E. Kindler and M. Weber. The Petri Net Kernel – an infrastructure for building Petri net tools. *Software Tools for Technology Transfer*, 3(4):486–497, July 2001.

[21] M. Laganeckas. A simulator for high-level Petri nets: Model-based design and implementation. Technical Report Masters thesis, IMM-M.Sc.-2012-101, DTU Informatics, Technical University of Denmark, Sept. 2012.

[22] OMG. Meta Object Facility (MOF) specification, version 1.4.1. Technical Report formal/05-05-05, The Object Management Group, Inc., May 2005.

[23] W. Reisig. *Elements of Distributed Algorithms — Modeling and Analysis with Petri Nets*. Springer, 1998.

[24] W. Reisig, E. Kindler, T. Vesper, H. Völzer, and R. Walter. Distributed algorithms for networks of agents. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets II: Applications*, volume 1492 of *LNCS*, pages 331–385. Springer, 1998.

[25] G. Rozenberg. Behaviour of elementary net systems. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of *LNCS*, pages 60–94. Springer-Verlag, 1987.

[26] P. H. Starke and H.-M. Hanisch. Analysis of signal/event nets. In *Emerging Technologies and Factory Automation (ETFA '97), Proceedings,* 6^{th} *International Conference on*, pages 253–257. IEEE, Sept. 1997.

[27] The Eclipse Foundation. The Eclipse platform. http://www.eclipse.org.

[28] P. Thiagarajan. Elementary net systems. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of *LNCS*, pages 26–59. Springer-Verlag, 1987.

[29] M. Weber and E. Kindler. The Petri Net Markup Language. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Petri Net Technologies for Modeling Communication Based Systems*, volume 2472 of *LNCS*, pages 124–144. Springer, 2003.

[30] M. Weber, R. Walter, H. Völzer, T. Vesper, W. Reisig, S. Peuker, E. Kindler, J. Freiheit, and J. Desel. DAWN: Petrinetzmodelle zur Verifikation Verteilter Algorithmen. Informatik-Bericht 88, Humboldt-Universität zu Berlin, Dec. 1997.

Index